SOSAP-TR-20A

July 1977

THE META DESCRIPTION SYSTEM: A System to generate Intelligent information Systems.

PART I: THE MODEL SPACE

by

C.V. Srinivasan

Dept. of Computer Science
Hill Centre, Busch Campus, Rutgers,
The State University of N.J.,
New Brunswick, N.J. 08903.

This work was supported by Grant DAHCIS - 73 - G6 of the Advanced Research Projects Agency.

THE META DESCRIPTION SYSTEM: A SYSTEM TO GENERATE INTELLIGENT INFORMATION SYSTEMS - PART I, THE MODEL SPACE.*

By Chitoor V. Srinivasan**

ABSTRACT: This paper discusses the architecture of a meta-system, which can be used to generate intelligent information systems for different domains of discourse. It points out the kinds of knowledge accepted by the system, and the way the knowledge is used to do non-trivial problem solving. The organization of the system makes it possible for it to function in the context of an expanding model space. The problem solving systems in the meta system communicate with the model space in the language defined for the domain. They have the capability to improve their performance, based on the knowledge gained from this communication. The meta-system provides a basis for the definition of the concept of machine understanding in terms of the models that the machine can build in a domain, and the way it can use the models.

---

* This work was supported by a grant from the Advanced Research Projects Agency (grant number DAHCIS-73-G6), of the Government of the United States of America.

**Department of Computer Science, Hill Centre, Busch Campus, Rutgers University, New Brunswick, N.J. 08903.

Table of Contents

1. INTRODUCTION

The Meta Description System (MDS) is a system for describing knowledge in a domain to a computer. MDS can be used to generate intelligent information systems, automatically from descriptions of knowledge in a domain of discourse. The domain could be diverse as for example, Medical Diagnosis [Irwin & Srinivasan 1975], Modelling Psychological Systems [Sridharan, Schmidt & Sridharan 1976, 1977] or Management Information System [Srinivasan 1974]. The major application of MDS has been so far in modelling psychological systems. A brief discussion of some of the features of MDS appears in Srinivasan [1973a, 1976f].

This paper presents informally the logical basis for the organization and operation of two of the major subsystems of MDS: Its MODEL SPACE, and the problem solving subsystem called, ASSIMILATOR,* which is responsible for the consistency of models in the MODEL SPACE. The ASSIMILATOR helps the MODEL SPACE assimilate assertions about models, making sure that the laws of a domain are not violated. It is also responsible for producing reasons that explain why certain assertions are accepted, others are not, and yet others are accepted conditionally on certain hypotheses. These reasons and hypotheses will constitute the systems own understanding

* This is a new terminology. This subsystem was being called INSTATIATOR-CHECKER in previous reports.

of the assertions. They also form the basis for all problem solving and intelligent activity in MDS. The reasons represent chunks of knowledge that may be used in a problem solving process to guide search.

The organization and operation of the MODEL SPACE and ASSIMILATOR are central to the MDS concept of knowledge itself, and its associated notions of understanding, description and use of knowledge. The objective of this paper is to present an operational view of this knowledge as it pertains to the MODEL SPACE.

The basic modelling concepts are presented in the context of a stylized domain of discourse: Transportation Systems and Problems like the MISSIONARIES & CANNIBALS (M&C), FARMER & SON (F&S), etc. This is done for pedagogical reasons.

In Section 1.1 below we shall give a brief outline of the central concepts in MDS, and introduce the MDS-paradigm for intelligent operation. We shall also establish the terminology used in the paper.

Implementation Status of MDS

There are now two partially implemented versions of MDS. One is MDS itself, which is implemented in INTERLISP. Only, the so called domain acquisition part of is now operational. This accepts definitions of domain knowledge in

the meta-language of MDS, and represents them in the model space in a form suitable for compilation and later use. Joel Irwin, John Ng and Tau Hsu participated in this implementation, together with the author. The second part of MDS, namely the code for the domain compiler, which derives from domain definitions procedures appropriate for the creation and maintenance of the model space, is now being written.

The other version of MDS is called AIMDS, Action Interpretation MDS. This is implemented in FUZZY [Le Faivre 1976]. There are differences between MDS and AIMDS in syntax and certain other definitional conventions. AIMDS is an interpretive version of a subset of MDS. It is now being used for modelling "belief systems".

1.1: Central Concepts and the MDS-Paradigm

1.1.1: Relational Systems, Description Language and Model Spaces.

The Model Space, M_T, * of a domain is central to the architecture of MDS, and its uses. The structures and processes in the model space are derived automatically by MDS from definitions of a Relational System, R_T. R_T itself is defined in the meta-language of MDS. R_T also forms the basis for the definition of the syntax and semantics of an elementary description language, L_T, which is used to describe, concepts and problems

---

* We shall throughout use the subscript T, as in M_T, to denote the prototypic domain, Transportation Systems.

(hereinafter called entities) in the domain.

Description Language, L_T

Every well-formed relation in R_T may occur as a phrase (literal) in L_T. Thus, in our domain, one may have phrases like "(p locationof h)", "(v cangoto p)" etc., where p is a PLACE, h say, is a HUMAN, m is an ANIMAL and v, a VEHICLE. L_T may also have, in addition, command phrases like (ASSERT (p locationof h)(h holding m)), (PROVE((ALL HUMAN h)(THERE-EXISTS PLACE p)(p locationof h))) etc. These are commands to one or more problem solving systems of MDS. "INSTANTIATE" is a command to the ASSIMILATOR. "ASSERT" and "GOAL" are commands to the DESIGNER, which is a goal-directed problem solver [Srinivasan 1976f, 1977c]. Whereas, ASSERT and GOAL may have preconditions, INSTANTIATE does not. "PROVE" is a command to the THEOREM PROVER (TP) [Srinivasan 1976c, 1977d**].

Sentences in L_T may be of two types: Declarative or Procedural. The sentential structure of declarative sentences is a variant of the sentential structure of the language of first order logic. Examples of declarative sentences appear in section 2. These are used to express the static laws of a domain: The laws of consistency that any given state of the model space should satisfy.

** In preparation.

The procedural sentences are used to express the dynamic laws: The laws of change in the model space. Every procedural sentence will have at least one occurrence of a command phrase. Examples of such sentences occur in section 2.5.9.

The semantics of L_T is entirely defined by the semantics of the relations in R_T. For the relations in R_T their semantics is defined by patterns of interactions that they may have in the model space: How the consistency of one relation may depend on the consistency of others. The forms of definitions of R_T, the rationale for their choice, their interpretations, and the assumptions on the control structure of the ASSIMILATOR that they entail, are all discussed in section 2.

Representations in the Model Space: Bundles.

The structure of R_T is also used by MDS to design a system of representations for models of entities in M_T. These representations are like the "frame" representations, proposed by Minsky [1975]. Each model is a data-structure representing a bundle of interconnected pieces of information, where each piece has its own substructure. Each bundle (model) may have a name, will have references to itself and its definition, and will contain a set of slots. Each slot is used to represent a property of the bundle.

Each property representation will itself consist of a set of slots : One each for the property value, reasons for the value, hypotheses, rules of transformations, and patterns of interactions with other bundles in the model space.

Each such bundle is a natural unit of knowledge in the domain, in the sense that, as a unit, it participates in all interactions with other such units in the model space. From each bundle one may obtain a view of the entire model space, as it pertains to the bundle. Also, the bundle is the focus of attention in all processing done in the model space. As we shall see, the bundle structure not only introduces a "modular" system design, but also, in a very real sense makes intelligent processing feasible, in practice.

The most significant aspect of MDS is that the bundle representations and all its associated processors for establishing and maintaining a consistent model space, are compiled automatically by MDS from schematic definitions of the structure and semantics of R_T. The compiled procedures are such that for every event in the model space the system can supply the reasons for the event, in the language L_T. The nature of this compilation process is briefly outlined in section 2.3.2.

The Model Space Logic

The model space works in a three-valued logical system: T(True), ?(Unknown) and NIL(False), T > ? > Nil, ~T = NIL and ~? = ?. The ability to instantiate schemas defined in R_T, update properties of bundles, and do model based reasoning and hypothesis generation, is primarily due to the use of the 3-valued logic. We shall use the phrase, Common-sense reasoning, to refer to the model based reasoning done by the ASSIMILATOR. Examples of this reasoning process are presented in section 2.5.6 through 2.5.8. The deductive mechanisms used for this process are defined in section 3.

1.1.2 The Problem Solving Systems of MDS

The ASSIMILATOR is the monitor for the model space. All events in the model space are initiated and directed by the ASSIMILATOR. It recognizes four kinds of commands: Instantiator Commands, examples of which we saw before; Recognition Commands, which are used to identify the class membership of a bundle; Comparison Commands, which are used to test for equality of bundles; and Retrieval Commands which are used to identify the bundle or bundles associated with a given name or phrase in L_T. The instantiator commands may present at a time a set of relations, all of which are to be instantiated in parallel. The assimilator is responsible to translate the relations to their associated representations, or modifications to representations, in the

model space. For example, to assimilate "(p locationof h)" it might be necessary to make several secondary changes in the model space, in order to maintain consistency. Thus, if h was holding m, then the location of m should also be set to p. Also, if h was initially at the place, g, then ~(g locationof h) should be made true in the new model state. The ASSIMILATOR can recognize and incorporate such secondary changes.

The ASSIMILATOR does not have the domain knowledge or the control structures necessary to plan and execute a sequence of actions in the model space. Thus, it would not know that h should have arrived at p using a VEHICLE, if such was the case. It is the role of the DESIGNER to recognize preconditions for actions and plan action sequences, that seek to achieve given goals. Thus, in response to (GOAL(p locationof h)) the DESIGNER might construct the sequence: For a vehicle, v,

(ASSERT (v holding h))
(ASSERT (p locationof v))
(ASSERT ~(v holding h)),

and present to the ASSIMILATOR the corresponding actions one by one. The reasons supplied by the ASSIMILATOR for the success or failure of the actions may be used by the DESIGNER in the planning process itself, to modify a

given plan. Thus, for example, the DESIGNER might receive the explanation "~(v location is locationof h)", as the reason for the failure of the assertion "(v holding h)". To correct the error the DESIGNER might choose another VEHICLE, v_1, which is at the location of h, or else have v brought to where h is. Also, the next time the DESIGNER makes an assertion like "(v holding h)" it would already make sure that v and h are at the same location. Thus, the DESIGNER would have learnt an elementary fact of the domain. The learned information would be summarized as a rule in the DESIGNER's own local state. Rules like this will be used by the DESIGNER not only to avoid repeating mistakes, but also to make the right choices, wherever feasible. Examples of these processes are discussed in Srinivasan [1976f, 1977c]. The logical basis that makes it possible to use reasons in this manner is established in section 3.

The DESIGNER will attempt to plan and achieve only goals that are specified to objects that already exist in the model space, or objects that are explicitly created, using the INSTANTIATE (or CREATE) commands. The DESIGNER does not have the control structure necessary to direct and sequence through a construction process to model predicate expressions. For example, the DESIGNER cannot prove an assertion like:

((ALL HUMAN h) (THERE-EXISTS PLACE p)(p locationof h)).

The THEOREM PROVER (TP) is used in MDS to prove assertions like the one above, assertions that hold universally true in a domain. The TP uses Gentzen's system of natural deduction. It uses the model space to discover the constraints that given assertions imply. The method of proof is like Beth's Semantic Tableaux method [Srinivasan 1976c]. The proof is attempted by seeking to build a model for a counterexample. Details of the TP are presented in Srinivasan [1977d]. They are briefly discussed in section 3.5.3.

These problem solving systems determine the scope of the system's understanding for a given corpus of domain knowledge. In some sense the concept of domain knowledge itself seems to exist only because of the existence of control structures that can do these various kinds of problem solving. The capability for understanding, exhibited by MDS in the context of these problem solving systems, forms the basis for language understanding in MDS. In the realm of description languages, L_T, has the status that "assembly languages" have in programming systems. Higher level description language may be defined and processed in MDS using the subsystem called, LINGUIST. The formalisms and processes of the LINGUIST will be

described in a future paper.

The LINGUIST is responsible to translate sentences to phrases, that ASSIMILATOR can understand. As shown in figure 1, the LINGUIST can use the DESIGNER and the TP in this process. The hypotheses supplied by the ASSIMILATOR will be used by the LINGUIST, as expectations in the context of a discourse. The reasons for contradictions in the model space will be used to seek alternate interpretations.

The collection of all reasons and hypotheses generated during the assimilation of a sentence will represent the system's own understanding of the sentence.

1.1.3: The MDS - Paradigm

The MDS-Paradigm is shown in figure 1. It consists of two parts: The domain definition part and the domain utilization part. In the domain definition part the meta-language of MDS is used to define R_T and L_T. These definitions are translated to representations in the model space by the Domain Acquisition System (This part of MDS is now operational). These representations of R_T and L_T are used by the domain compiler to generate code for the creation and maintenance of models in the model space. (This part is now being coded).

In the domain utilization part, sentences in L_T are

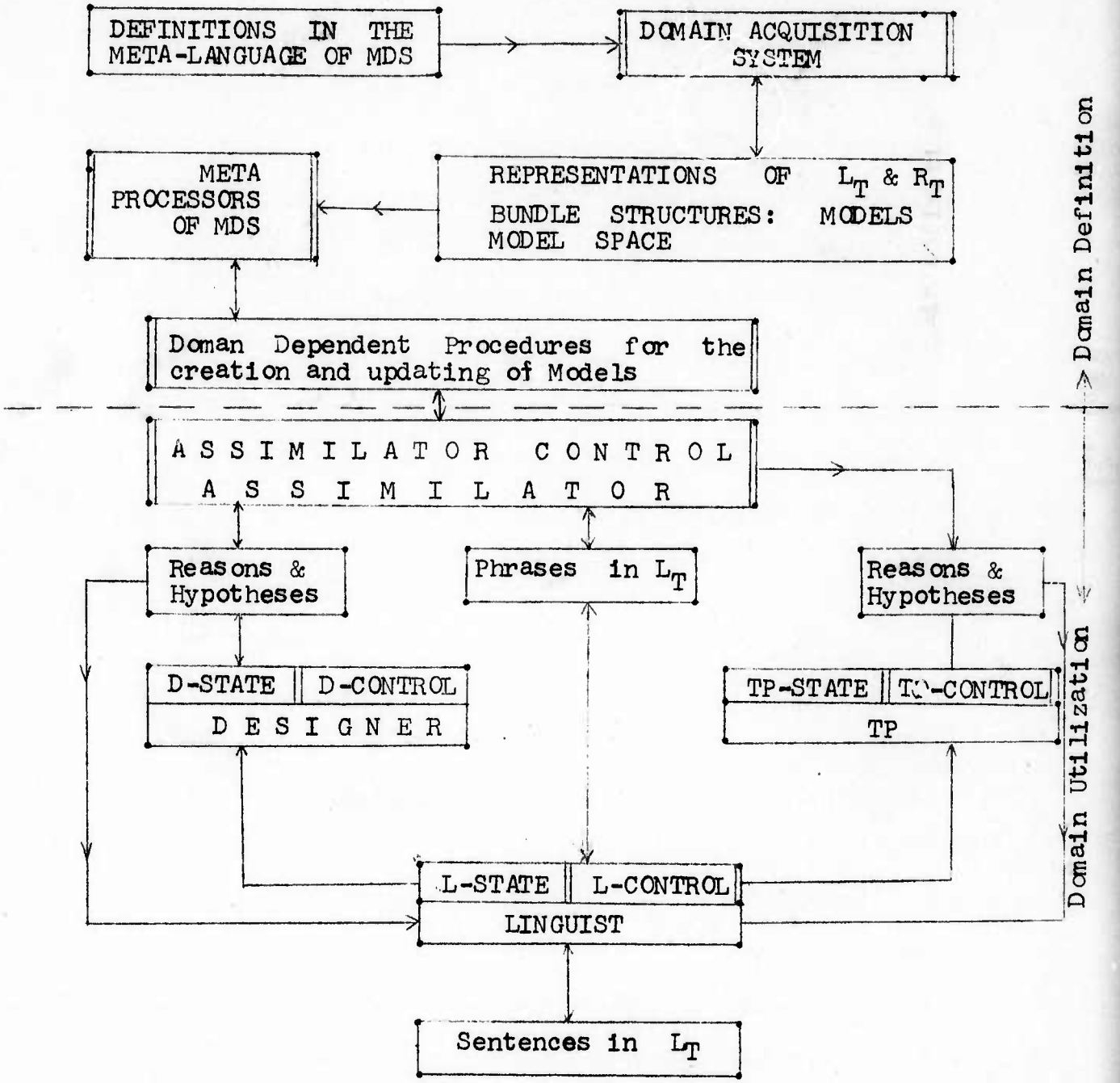

Figure 1: The MDS - Paradigm

received by the LINGUIST, which is responsible to translate them to phrases in L_T that ASSIMILATOR can understand. In doing this translation, the LINGUIST may make use of the DESIGNER and the TP. Each one of these problem solving systems has its own local state. These local states are kept updated by the respective controls of the problem solving systems, with information received from the ASSIMILATOR, namely the reasons and hypotheses generated by the ASSIMILATOR in response to the phrases at its input. The problem solving systems use the information in their respective states to guide their own activities. This forms the basis for learning in MDS.

The local state of a problem solving system may be viewed as a model of the system's own past activities. Such models may again be themselves described schematically in the meta-language of MDS. Thus in MDS, one may specify how a problem solver should model its own activities. The modelling schemas for the DESIGNER'S problem solving state are discussed in Srinivasan [1977c]. For a given problem solver like the DESIGNER its modelling schemas may be domain dependent, or within a domain it may depend on types of problems. Thus domain knowledge may appear in bundles in MDS at various levels of the system's activities: At the level of the model space it may appear as the data-base for a domain. At the level of the LINGUIST, it may appear as definitions of the syntax

of L_T or as decision processes associated with LINGUIST. In each problem solving system the bundles may appear in the representation of the problem solving states and in the decision processes of the problem solver.

The logical basis for learning and commensense reasoning within this paradigm is established in section 3.

1.2. Relationship to other systems

MDS incorporates in its organisation many of the important concepts that have been developed in AI-systems over the last two decades:

"Means-end analysis" [Newell, A, et. al. 1959] is used in model-space updating, and in problem solving by DESIGNER; "Theorem proving" is used to establish general assertions in the domain; "Procedural specifications of knowledge" is used [Hewitt 1972, Winograd 1975] to define the dynamic laws of the domain, the laws of change in the model space; "pattern-based invocation" of procedures is used by DESIGNER; "declarative specifications of knowledge" in first order logic is used to describe the static laws of a domain, the laws of consistency that a model space should satisfy; knowledge in the model space is represented in "frame" like bundles, which behave like molecules in interactions with each other; finally, relational systems are used as the basis for defining model spaces and

languages. MDS provides the logical frame work and an architecture in which these concepts function together.

MDS does not seek to define yet another programming language or seek to contribute to programming techniques. MDS is a problem solving system, which can generate programs from definitions of knowledge in a domain. It proposes a view of knowledge, and provides a formalism for defining knowledge. In defining this knowledge a user need not concern himself with possible interactions between components of his definitions, and provide procedures to process such interactions. As we shall see in section 2.5.7. MDS itself can derive from the definitions the procedures necessary to anticipate and process interactions between models. Thus, the definitions themselves are modular. We shall see examples of these definitions in section 2.

MDS makes a clear distinction between the model space and its control structures, and the control structures of the various problem solving systems. The representations and processes in the model space are completely independent of the problem solving systems, like DESIGNER, TP and LINGUIST. Because of the commonsense reasoning paradigm each problem solver is able to get from the model space new combinations of domain knowledge specific to its own needs. Using these chunks of knowledge each problem solver may develop its

own specific views of the model space. The ability and efficiency of these interactions between the problem solvers and the model space is limited only by the primitives -- descriptive relations -- available in the model space. Since the relational system determines the description language of a domain, this is the same as saying that the problem solving efficiency is dependent on the concepts expressible in the description language. Thus, MDS brings to focus the knowledge representation issue as an issue of language design, and not as an issue of program design or data-structure design.

Major innovations in the MDS architecture are:

(i) the use of relational systems as the basis for the design of model spaces and languages;

(ii) The formalism used to describe knowledge, and the control structures that can use the defined knowledge to do useful work;

(iii) The architecture of the model space in 3-valued logic and the commonsense reasoning paradigm.

We present in this paper, the logical foundations of this architecture, and establish the basis for using MDS to do a variety of problem solving activities, activities such as assimilation, goal directed planning and problem solving, theorem proving, recognition, understanding, etc. Details of these problem solving activities themselves will be discussed in future papers.

II. MDS MODEL SPACE AND THE ASSIMILATOR

2.1 Introduction

In this chapter we shall introduce the logic, architecture, methods and uses, and forms of definitions of the MDS model space. We shall define R_T, and discuss interpretations given to the various components of the definitions in the model space. We shall present examples of L_T, and structures and processes in M_T, that are implied by R_T.

The definitions of R_T will contain three components; structural, sense and transformational. For each component we shall discuss the definitional forms, and the rationale for the choice of the forms shown. We shall also point out the assumptions on the ASSIMILATOR control that they entail.

We shall see examples of commonsense reasoning in the model space and the role it plays in the maintenance of consistency. The logic of this reasoning process and the associated deductive mechanisms are defined in chapter III.

2.2 Knowledge Representation: Focus on Objects and classes

There are two extremes in knowledge representation: One may be called Operator-based and the other object-based. In operator-based representations objects are treated as uninterpreted formal entities, which may appear as components of a

model-state. A model-state itself is described in terms of how one might arrive there, starting from one or more distinguished initial states, by applying one or more sequences of transformations (Operators). Examples of such representations appear in Algebra, Group Theory and in certain game playing systems. They are useful in situations where total knowledge is available, and the objects involved show certain closure properties with respect to the operators.

In MDS the bias is predominantly towards object-based representations. Here operators and functions are characterized in terms of how they affect properties of classes of objects over which they may operate. The representations focus on interactions between properties of objects. This kind of representation leads naturally to the so called "frame systems". There is no notion of model-state. The model, in fact, is likely to be always incomplete. Object based representations have several advantages. We shall discuss them in the last section of this chapter. It is useful to first develop an intuitive understanding for the nature of representations in the model space and their implications.

2.3. The Structure of Descriptions

The constructs discussed below cover most of modelling concepts in MDS. Certain aspects pertaining to the specifications

of properties of relations, relation hierarchies, and definitions of tuple and function schemas, as well as meta-schemas are not discussed. These are defined in Srinivasan [1976e]. In what follows, we shall use square brackets, "[" and "]" to enclose tuples, chain brackets, "{" and "}" to enclose sets, and parentheses, "(" and ")" to enclose collections. We also use parentheses to delimit relational forms and expressions in L_T. So also, we use, at times, the square brackets in expressions in the way INTERLISP uses them, to indicate automatic closure of parentheses in nested expressions. But these should not cause any confusion.

2.3.1. Description Schemas and Templates

We shall use descriptive relation names like "locationof", "cangoto", "candrive", "holding", etc. to describe properties of objects like PLACEs, VEHICLEs, HUMANs, etc. For each such relation name we shall specify the classes of objects which it may relate. This will define the forms of the literals that may appear in L_T. Thus, we shall say that

[S1] (PLACE Locationof ITEMS)

is a description schema associated with "locationof". A phrase like "(x locationof s)" is said to be dimensionally consistent only if x is a PLACE and s is an instance of ITEMS, say s= $(x_1\ x_2 \ldots x_n)$. Here, $(x_1\ x_2\ \ldots\ x_n)$ is called a collection, also at times, called a list. We will say

$$(x \text{ locationof}) = (x_1\ x_2\ \ldots\ x_n), \tag{2-1}$$

or write this as functional,

$$(x \text{ locationof } (x_1\ x_2\ \ldots\ x_n)) \tag{2-2}$$

with the interpretation,

$$(x \text{ locationof } x_1) \wedge (x \text{ locationof } x_2) \wedge \ldots \wedge (x \text{ locationof } x_n) \tag{2-3}$$

Thus, by convention, relations distribute over collections. We shall constrain the elements of ITEMS to be instances of HUMAN, ANIMAL, VEGETABLE or VEHICLE. A given instance of ITEMS may have an arbitrary number of instances of these elements. In MDS, this is specified by the schema:

[S2]: (ITEMS elements (HUMAN ANIMAL VEGETABLE VEHICLE)),

where "elements" is a distinguished relation of MDS: This relation may appear only with classes that are sets or collections (lists). Collection and sets in the model space should be contrasted with nodes like PLACE, HUMAN, etc. Every node is an individual. "(x elements)" is dimensionally inconsistent for a node, x.

For every relation name, r, we shall define an inverse, r' such that

$$((\forall x)(\forall y)(x\ r\ y) \Leftrightarrow (y\ r'\ x)) \tag{2-4}$$

In our domain, "locationof" and "location" are inverses of each other. To satisfy (2-4) we may now define either,

[S3] (ITEMS location PLACE),

or for each possible element of ITEMS one may define its associated schema, as follows:

[S4]: (HUMAN location PLACE)

[S5]: (ANIMAL location PLACE)

[S6]: (VEGETABLE location PLACE) and

[S7]: (VEHICLE location PLACE).

We shall choose the second alternative. In MDS, a schema like [S4] is interpreted to mean that for every HUMAN, h, there is only one PLACE, p, such that (h location p). This is because PLACE is a node. Thus, it would be dimensionally inconsistent to say (h location (p^1 p^2)) or (h location NIL)

Another kind of schema definition occurs in the case of the relation name, "heldby", which is the inverseof "holding". An object can be heldby a HUMAN or a VEHICLE. This may be indicated by the schemas:

[S8] (ITEMS heldby HUMAN\VEHICLE)

[S9] (VEHICLE holding ITEMS)

[S10] (HUMAN holding ITEMS)

Here the phrase "HUMAN\VEHICLE" is interpreted as (ONEOF HUMAN VEHICLE). In our domain an object can be heldby only one object.

Structural schemas like S1 through S10 are declared in MDS by using devices called, Templates. Each template will define all the schemas associated with a given class of objects in the domain. The templates for PLACE and ITEMS are shown below.

Here. "TDN:" is the "Template DefinitionN" command in the existing implementation of MDS. The words "RN" and "$L" associated respectively with PLACE and ITEMS, in the definitions below are flags that indicate special representation or interpretations associated with the templates. The flag "RN"

TEMPLATES FOR PLACE AND ITEMS:
[TDN: (PLACE RN)(locationof ITEMS)]
[TDN: (ITEMS $L)((heldby V) HUMAN\VEHICLE holding)
(elements HUMAN VEGETABLE ANIMAL VEHICLE)]

defines PLACE to be a "regular node" template. It is a node, and it is regular in the sense that every instance of PLACE should have a name. Names are used in the model space and in L_T to denote objects (models). A model without a name is called a dummy model. The flag "$L" associated with ITEMS defines items to be a "dummy list". Thus, every instance of ITEMS is a list (collection), and an instance, say $(x_1\ x_2\ \ldots x_n)$, may not have any name associated with it. The only way of referring to such an instance in L_T would be via an associated relation, like say (p locationof), where (p locationof) might be equal to $(x_1\ x_2\ \ldots\ x_n)$.

The form "(heldby V)" in the ITEMS template, declares "heldby" to be a variable relation, in the context of ITEMS: There may be items, t, such that (t heldby) = NIL. This template also declares "holding" to be the inverse of "heldby". If no inverse is specified, as in the case of "locationof" in the PLACE template, then the inverse is obtained by deleting (concatenating) the suffix "of" from (to) the indicated name.

Thus, the inverse of "locationof" will be "location".

Templates define the dimensionality of relations: What classes a relation name may relate. They also specify the description structure of a class: What relations are used to describe an instance of a class. In addition, as we shall later see, they implicitly define a representation scheme for storing descriptions of instances of a class in the model space. They also, of course, define a language to refer to the components of such descriptions.

The templates for the various classes in our domain are shown in Table I. It is suggested that the reader get familiar with the various classes defined in Table I, and their respective description structures. Some of the templates in this table contain labels, CC1, CC2, etc., associated with certain relation schemas. These labels indicate the presence of Consistency Conditions (CC's) associated with the respective schemas. For an instance x of X, the CC associated with (x r y) will constrain the instances, y of Y, which may appear as values of the phrase (x r) in the model space: (x r y) can be true if and only if the CC associated with (x r y) is satisfied by x and y. The forms, interpretations, uses, and properties of CC's are discussed in the ensuing sections of this chapter. The CC's define the sense of the relations in the model space. Their forms and interpretations, in fact, establish the basis for the use of MDS as a meta-system to generate intelligent information systems. We shall enter into a discussion of CC's after introducing some of the

TABLE I: Templates for the domain of Transportation Systems

1. [TDN: (HUMAN RN) (type HTYPE)
(candrive VEHICLES canbedrivenby)
(compatiblewith ITEMS compatiblewith, CC1)
(holding ITEMS heldby)
(location PLACE].

2. [TDN: (VEHICLE RN) (capacity INTEGER)
(canbedrovenby HUMANS)
(holding ITEMS - CC2)
(location PLACE)
(cangoto PLACES canbereachedby)].

3. [TDN: (ANIMAL RN) (type ATYPE)
(compatiblewith ITEMS - CC3)
(location PLACE)].

4. [TDN: (VEGETABLE RN) (location PLACE)
(compatiblewith ITEMS - CC4)].

5. [TDN: (PLACE RN) (locationof ITEMS - CC5)]

6. [TDN: (ITEMS $L)
((heldby v) HUMAN\VEHICLE - CC6))
(elements HUMAN VEGETABLE ANIMAL VEHICLE)]

7. [TDN: (HTYPE RN)(typeof HUMANS)]

8. [TDN: (ATYPE RN)(typeof ANIMALS)]

commands of the ASSMILATOR, the nature of representations in the model space, some of the other definitional facilities in MDS, and the concepts of domain compilation.

2.3.2 The ASSIMILATOR Control

2.3.2.1: The Domain Compiler and associated Facilities in MDS.

A template, X, may be used as the basis for designing a system of representation for storing descriptions of instances of X in the model space. The form of this representation may depend on the characteristics of the storage medium. Thus, the representation scheme for storing descriptions in a "secondary store" would be different from the representation scheme for a "primary store". We shall here postulate the availability of functions Pi, i = 1,2,...,m,

$$Pi : \{<templates>\} \xrightarrow{1-1} \{<datatypes>\} \qquad (2.5)$$

that map templates uniquely to their corresponding datatypes in the storage medium, i. These functions will be a part of the, so called, domain compiler.

MDS has several definitional facilities that assist the domain compiler to design representations and produce efficient compiled code. Some of these are reviewed below.

The flags "RN" and "$L", that we mentioned before, are directly relevant to the design of representations. Some of the other template flags currently used in MDS are given below:

RT and $T: Regular and Dummy Tuple schemas. These are used to define n-ary relations for $n \geq 1$. In the examples discussed in this paper only binary relations are used. We find that in most domains binary relations are the ones that are used predominantly.

RF and $F: Regular and Dummy Function schemas. These are used to define functions which may be declared as part of a relation definition schema. Thus one may have a schema of the form (X r F) where F is a function template. In this case, for an instance x of X, the value of (x r) will be obtained by executing F on arguments which may themselves be determined by x. For examples of use of Function Scheme see Irwin & Srinivasan [1975]. Function Schemas may be used in MDS to define structures similar to semantic nets.

TI, T#, TS, etc: These are the various Terminal Templates, Terminal Integer, Terminal Number, Terminal String, etc. In the current implementation of MDS (which is in INTERLISP), all INTERLISP datatypes are also available as templates in MDS, Also, every datatype used in the implementation of MDS itself is available to MDS as a

template. Thus, there are templates for representing templates, for representing names, relations, constraints, actions, models, sets, etc. some of these are discussed in Appendix II.

MN,ML,MT,MF,etc.: These refer to various kinds of meta-templates. An instance of a meta-template is itself a template. Thus, an instance of Meta Node (MN) template will be a Node template. It could be a Dummy Node or a Regular Node. For examples of uses of meta-templates see Irwin & Srinivasan [1975].

One may also associate flags with description schemas of the form (X r Y). These flags fall into the following categories:

(i) Flags that define relation properties.

Properties like transitively, reflexivity, etc. in the context of an [X r] may be declared to the system by using relation flags. These declared properties are used by the domain compiler in generating codes associated with [X r].

(ii) Flags that define interpretations for Functionals

In the schema (X r Z), where Z is a collection with (Z elements Y), the normal interpretation given to (x r (y_1 y_2...y_n)), where x is an instance of X and (y_1 y_2...y_n) is an instance of Z, is

$$(x\ r\ (y_1 \ldots y_n)) \Leftrightarrow (x\ r\ y_1) \wedge (x\ r\ y_2) \wedge \ldots \wedge (x\ r\ y_n) \ldots (2.6)$$

However, if the flag "S' is associated with [X r] then the collection $(y_1\ y_2 \ldots y_n)$ will be interpreted as a set, $\{y_1\ y_2 \ldots y_n\}$, in the context of (x r): $(X\ r\ \{\ y_1\ y_2 \ldots y_n\})$ does not necessarily imply $(x\ r\ y_i)$ for $i = 1,2,\ldots,n$. Thus, sets and collections have different interpretations in the model space. Similarly, one may use the relation flag "B" in the context of [X r], if (X r Y) is true and Y is a tuple-template. In this case, instances of Y will be interpreted as Bags in the context of (x r).

(iii) Flags that specify storage control

Normally, for every (x r), if (x r) is dimensionally consistent, then the value y such that (x r) = y is stored in the model space in the form of representation of the relation (x r y). However, one may specify by the use of the, so called, dummy flag, that the value of (x r) is not to be stored. In this case, every time (x r) is requested its value will be computed using the function, CC's and transformations associated with [X r]. The symbol "$" is used for the dummy flag. If ([X r] flag $) and (X r Y) are true, then ([Y r'] flag $) is implied, where r' is the inverseof r. We shall see a use for the dummy flag in section 2.5.9.

Other storage control flags may be defined to specify storage of meanings, and action interpretations. There is

also a flag called the prompt flag*. The symbol "!" is used for this flag in the current implementation of MDS. If ([X r] flag !) is true then, every time a new instance x of X, is created the appropriate value of (x r) should also be instantiated. This may, at times, necessitate the system to prompt a user to supply a value for (x r). Examples of use of this flag appear in Irwin and Srinivasan [1975].

(iv) Protection Flags

These may either explicitly indicate the protection associated with the access and updating of relation values, or present conditions under which such access is permissible. A common protection flag is the "*" flag, which indicates that the value of an (x r) cannot be changed during a problem solving process.

(v) Context or Intention flags

In MDS there may exist simultaneously several model spaces for a given domain, each associated with a different context. So also for an object, x, there may exist several models of x, each associated with a different context. One may explicitly associate with a description schema, the model context, in which the constraints associated with the schema are evaluated.

In the current implementation of MDS there are facilities for extending the repertoir of flags used with the templates

* The use of this feature was suggested by Sridharan.

and relations. Each flag definition may itself be controlled by flag templates. The definitions of these flags as per the flag templates will enable the domain compiler to incorporate these flags into the compiled code.

For each (X r Y) one may also define in MDS an associated action, called the Anchored Transformation Rule (ATR). This rule will be invoked when necessary during the instantiation of an (x r) for an instance, x, of X. We shall see examples of the use of ATR's in section 2.5.9.

It should be noted that "instance" and "instanceof" are distinguished relations in MDS, which are associated with every template. Thus for a template, X, one may have CC's and ATR's associated with [X instance] itself. These will be invoked every time one attempts to create a new instance of X. Both the CC's and the ATR's may be used during the domain compilation process to produce efficient compiled codes for a domain. We shall discuss the details of this compilation process in another report.

2.3.2.2: Examples of Representations, Access Functions and the Basic Commands of the ASSIMILATOR.

The representation for an instance of PLACE might be

To definition of PLACE	self-reference	Locationof	elementof

and that of ITEMS,

To definition of ITEMS	self-reference	heldby	elements	elementsof

By convention, every data type may appear as the element of one or more collections. Thus, we have the "elementof" relation pointer appearing in both the data types above. Since ITEMS is a collection, it also has a pointer for the "elements" relation. The first field in every data type will point to the representation of the template that is associated with the data type. The second field is a pointer to the instance itself. The remaining fields correspond to the relations defined in the template associated with the data type. For each relation its associated field will point to a, so called, descriptor unit (DESUNIT). The DESUNIT will have slots for the value of the relations, reasons, hypotheses, etc., as mentioned before in the description of bundles. Each bundle in the model space will correspond to the representation of the model of an entity in the domain. A more detailed discussion of these representations appear in Srinivasan [1977c].

At this point let us take note of the basic commands of the ASSIMILATOR. For each data-type (template) there will be four associated classes of commands:

[C1]: Recognition Commands
[C2]: Creation Commands, and
[C3]: Comparison Commands.
[C4]: Retrieval Commands.

These are briefly described below. Let MACC be the accumulator of the Model Space. We will use the symbol "@" to denote the Contents of MACC. We shall refer to @ as the anchor of the model space. It is the current object of focus in the model space. Where convenient we shall use symbols $@_X$, $@_Y$ etc. for

pedagogical reasons to indicate anchors that are instances of X, Y etc. respectively. For all the commands described below, the anchor is one of the argument of the commands.

[C1]: (DTYPE d) = T,?, NIL.

The result is T if the datatype of @ is d, is ? if it is unknown, else NIL.

[C2]a. Instantiate Template

(IT d m) = x or NIL.

The result x is a pointer to the newly created instance of data type, d, with name, m, assigned to it, if m is given. x is put in MACC if the instantiation is successful. Else, MACC will contain NIL.

[C2]b. Instantiate Relation

(IR r y) or (IR x) = T, ? or NIL.

This will attempt to make (@ r y) true -- if y is not given it will attempt to find the appropriate y. The command will succeed only if there is no resultant contradiction in the model space. The result of this operation is put in the MTEST register of the ASSIMILATOR.

[C2] (c) Instantiate Relation Negative

(IRN r y) or (IRN r) = T, ?, NIL.

This is similar to IR but attempts to make ~(@ r y) true.

Corresponding to [C2] (b) and [C2] (c) one may also have commands FR and FRN (Force relation and Force relation negative).

These would attempt to force (@ r y) -- or ~(@ r y) -- by modifying the model space appropriately, if necessary.

[C3] (EQUALS x) = T, ?, NIL

This checks (@ = x). The result is stored in MTEST.

[C4] Retrieval Commands

The above commands may be compiled for each domain, from the domain definitions. For each datatype the domain compiler will also produce codes for the access functions, (@ r), for obtaining the value of the field associated with the relation r in @. If (@ r) is unknown, then the access functions may invoke the cc's and ATR associated with (@ r) to find its value. If (@ r) is dimensionally inconsistent then the value of (@ r) = NIL. Similarly, one may also check the truth value of (@ r y) using the access functions.

The ASSIMILATOR structure, presented above, has the form of a machine. The commands are like machine commands. This is deliberate. As discussed below, we do visualize a machine control, in which the domain dependent processors are micro-programmed, and the basic ASSIMILATOR control invokes and executes them to manage the model space.

The nature of algorithms for some of these processors, and the associated data organizations are discussed in section 2.5. They are part of the forms and interpretations of the description schemas, templates, cc's and ATR's.

2.3.3: The Compilation Process

We shall assume the availability of an assembly language for the ASSIMILATOR with Commands of the form:

(INSTANTIATE (x r y) ~(x_1 r_1 y_1) ... (x_n r_n y_n))
(x r), (x r y), (EQUALS x y), (NTYPE x d), etc.

The assertions in the INSTANTIATE command are to be assimilated in parallel. The compilation of a command like this would involve four steps:

STEP (i): Determination of all structures in the model space which need be changed in order to accept the given collection of assertions. This is determined by the structural knowledge of a domain. For each assertion, (x r y), depending upon the classes of x and y its associated structures may be directly compiled from the templates of x and y, and the relation flags associated with r. All the structural changes, derivable from the given assertions using the description structures of the objects involved, will be hypothesized to be true in this step.

STEP (ii): This step determines all the consistency conditions associated with the hypothesized structural changes, and implied by the interactions of the hypotheses with other relations in the model space. Each one of these conditions (or reasons associated with the conditions) are evaluated for appropriate bindings of the free variables. The hypotheses may be accepted only if none of the conditions evaluate to NIL. This evaluation will also produce the combined reason

for the acceptance, conditional acceptances or rejection of the hypotheses.

Every one of the consistency conditions involved in such a check may be compiled. Also, the procedures necessary to identify relation interactions caused by the hypotheses may be derived and compiled from domain definitions. The details of this checking process are discussed in sections 2.5.7 through 2.5.9.

STEP (iii) If the checks evaluate to T or ? or NIL then the transformations associated with each asserted (x r y) for its associated truth value are executed. This might successfully terminate the assimilation process, and return the resultant reasons for the success. Or, it may terminate the process with the truth value ? and an associated hypothesis for the acceptance of the assertions. Or else, it may produce the truth value NIL, indicating the presence of a contradiction in the assertions, and go to STEP (iv). In this case, of course, the reasons for the contradictions will also be made available.

All the transformation rules may be compiled from the domain definitions. If no transformations exist then this step will be skipped.

STEP (iv) This step is used only if a contradiction has been recognized. The system would then attempt to eliminate the

reasons for the contradictions by proposing possible secondary changes to the model space. The analysis used for this purpose may also itself be compiled. This analysis is based on the reasons for the contradiction, and certain definitional entities called Focus Lists that are associated with the relations involved.

The details of this updating process are discussed in section 2.5, in the context of specific examples. The access functions for

$$(x\ r) = (y \mid (x\ r\ y))$$

and $(x\ r\ y) = T, ?$ or NIL

are compiled from the data-structures, cc's and ATR's associated with each templates. Also, for each template the EQUALITY checking routines for instances of the template are compiled from the domain definitions. The details of this compilation process will be discussed in a future report.

The ASSIMILATOR itself is thus a pure control structure, which would invoke the above mentioned compiled procedures where necessary to execute the commands received by it. For a given domain, with well understood knowledge representation schemes, all these domain dependent procedures may be compiled into "micro-programs" from given domain definitions. The control structure of the ASSIMILATOR is different from the structure of the "execution control", we see in all Von Neumann machines. It seems, for intelligent operation both execution control and assimilation control are essential. Details of the assimilation control are discussed in [Srinivasan 1977d].

2.4 Consistency Conditions or Sense Definitions

2.4.1 The Nature of Constraints: Some Examples

The schema [S1] does not specify all the restrictions associated with what can be at a given PLACE. Not any combination of items may be at the location of a given PLACE. In our domain we would like the following to be always true:

```
[(Vx)(Vy)(PLACE instance x)
 ((HUMAN instance y) V (VEGETABLE instance y)
 V(ANIMAL instance y) V (VEHICLE instance y))
   =>
((Vz)(x locationof z)(z holding y) ⇒
                      (x locationof y))
((Vz)(x locationof z)(x locationof y) ⇒
                      (SAFE y z))] ...          (2.7)
```

All the literals in (2.7) are dimensionally consistent with respect to the definitions in Table I. The predicate SAFE is as yet undefined. (2.7) asserts that if y and z are at the same place x, then (SAFE y z) should be true, and if z is holding y, and z is at x, then y should also be at x. The definitions of SAFE may be problem dependent. For the M&C problem one may have,*

```
M&C-SAFE
[(Vx)(Vy)(SAFE x y)  ⟺
 ((Vs)(Vp)(s instanceof ITEMS)
  (p instance of PLACE)
  (p locationof s) ⟹
     (((#OF MISSIONARY s)≥(#OF CANNIBAL s))
       V ((#OF MISSIONARY s) = 0)] ...          (2.8)
```

* We assume implicit conjunction between parenthesized forms.

Here, (#OF x y) is a function that returns the number of items of type x in a collection y. For the domain F&S, the definition of SAFE might be:

F&S-SAFE
(∀x)(∀y)(SAFE x y) ⟺
((x compatiblewith y) V
((∀p)(p instanceof PLACE)(p locationof (x y)) ⇒
((∃h)(h instanceof HUMAN)
(p locationof h)] ... ... (2.9)

In this case if x is not compatiblewith y then a HUMAN is required to be at the same PLACE as x and y. The constraint for (x compatiblewith y) in the case of HUMANs is shown below:

[(∀x)(HUMAN instance y) ⇒
((∀y)(x compatiblewith y) ⟺
(((∀t)(x type t) ⟺ (y type t))V
~(y instance of HUMAN]... ... (2.10)

Similar definitions for this relation, for other classes of objects, are shown in Table III. For a given HUMAN, h, (2.10) may be used to find all y such that (h compatiblewith y) is true: (y|(h compatiblewith y)). However, for a given PLACE, p, (2.7) cannot be used to find (y|(p locationof y)). But, if a candidate y is supplied then (2.7) may be used to check whether (p locationof y) could be true for the candidate. We shall call constraints like (2.7), <u>declarative constraints</u> (not to be confused with declrative descriptions of knowledge). Constraints

like (2.10) are called imperative constraints. A formal definition of these concepts is given in section 2.5.5.

The forms of (2.7) and (2.10) are not quite satisfactory for the purpose of modelling in terms of object-based representations. We shall state the constraints in a form, that would facilitate the realization of the goals discussed in the next subsection.

2.4.2. What should Sense Definitions do?

Objective [Ob1]: Ensure Model Space Consistency

In the three valued logical system, we shall require of the model space only a week state of consistency: It should be at all times contradiction free. Thus, the model space may contain relations whose truth values are unknown. This may, at times, result in the following kinds of situations: Consider the chains

(a) $(x\ r\ y) \Rightarrow (x_1\ r_1\ y_1) \Rightarrow \ldots \Rightarrow (x_n\ r_n\ y_n)$

(b) $(u\ t\ v) \Rightarrow (u_1\ t_1\ v_1) \Rightarrow \ldots \Rightarrow \sim(x_n\ r_n\ y_n)$

If the truth value of $(x_n\ r_n\ y_n)$ is unknown (?) in the model space, then it can accept the assertion $(x\ r\ y)(u\ t\ v)$ -- we assume implicit conjunction. This is because, $\sim ? = ?$ and accepting $(x\ r\ y)(u\ t\ v)$ would not cause any contradiction. We shall, however require that the model space be such that, at a later time, if $(x_n\ r_n\ y_n)$ is asserted, the latent contradiction should surface.

Objective [Ob2]: <u>For each (@ r) find if possible one of the following</u>:

(a) The y such that (@ r y) is true if such a y exists in the model space.

(b) The candidates (y_1 $y_2 \ldots y_n$) for one or more of which (@ r y) may be true.

(c) The constraints specific to @, that characterize all y such that (@ r y) is true.

<u>Objective</u> [Ob3]: <u>Give Reasons</u>

If (@ r y) = T, ? or NIL then identify and express the reasons for this in L_T. This is the most important requirement. The satisfaction of this objective makes it possible to do problem solving in MDS.

<u>Objective</u> [Ob4]: <u>Anticipate Interactions</u>

For each (@ r y) identify the specific interactions that take place in the model space with other relations that may exist in the model space.

<u>Objective</u> [Ob5]: <u>Avoid Combinatorial Explosions</u>

In seeking to satisfy [Ob1] through [Ob4], and in using the model space to solve problems, it should be possible to <u>specify starategies</u> and learn <u>rules</u> that contribute to minimizing combinatorial explosions.

We shall present below the elements of a system architecture in which all the above objectives may be realized.

2.5. Representations and Uses of Constraints:

2.5.1. Use of Bounded Quantifiers

The weak definition of model space consistency makes it sufficient to check for each (x r y) the relevent constraints only over the objects and relations that actually exist in the model space, at the time (x r y) is asserted. It is not necessary to resolve hidden contradictions because of unknown quantities. Thus all quantifications in our constraints will be bounded, and general predicate expressions may be reduced to conjunctions and/or disjunctions of propositions, whose truth values may be directly tested in the model space.

Further, one may notice that variables range only over specified classes of objects in the model space. Thus in (2.7), x ranges only over PLACES, and y ranges only over what can appear as elements of ITEMS. To take advantage of these categorized variables we shall modify the language of constraints indicating explicitly, where feasible, the range of each quantified variable. We shall use

"(<classname> x)(Px...)"

to denote

"((∀x)(<classname> instance x) ⇒ (Px...))",

and use

"((SOME <classname> x)(Px...))"

to denote

"((∃x) (<classname> instance x)(P x...))",

where(P x...) is predicate expression in which x occurs free. Where appropriate we shall also use form, "(<class1>/<class2>/../<class> x)" to denote a range that extends over a disjunction of classes, and forms "(<class> x y)" for "(<class> x)(<class> y)", and "(SOME <class> x y)" for "(SOME <class> x)(SOME <class> y)".

2.5.2 The Use of Relation Paths

We will use ":" to denote relation concatenation, and call phrases "$r_1:r_2\ldots:r_n$", relation paths, We shall use "(x $r_1:r_2$ y)" to denote "((x r_1) r_2 y)". In view of (2-6) and the convention,

$$(x\ r) = (z\ |(x\ r\ z)),\ \ldots \tag{2.11}$$

it follows that

$$(x\ r_1:r_2\ y) \Leftrightarrow ((\forall z)(x\ r_1\ z) \Rightarrow (z\ r_2\ y)). \tag{2.12}$$

If r_2' is the inverse of r_2, and in the structural description both (x r_1) and(y r_2') are constrained to be nodes, or collections of equal cardinality then

$$(x\ r_1:r_2\ y) \Leftrightarrow((\forall z)(x\ r_1\ z) \Leftrightarrow(z\ r_2\ y))\ .. \tag{2.13}$$

For a relation path $r_1:r_2:\ldots r_n$ its inverse path is $r_n':r_{n-1}':\ldots r_2':r_1'$. Using these conventions we may now rewrite (2.7) and (2.10) as follows:

```
[(PLACE p)
 (HUMAN/VEGATABLE ANIMAL/VEHICLE x y)
 ((p locationof (x y)) => (SAFE x y))
 (x heldby:location:locationof    x)]  ...      (2.14)

[(HUMAN h)(y)(h compatiblewith y) <=>((h type:typeof y) V
     ~(y instanceof HUMAN)]                ...      (2.15)
```

2.5.3. The Use of Definitional Anchors

With every constraint we shall associate a distinguished relation name, called the anchor relation of the constraint. The anchor relation of (2.14) and (2.15) are "locationof" and "compatiblewith," respectively. We shall anchor the constraint itself at the, so called, definitional anchor, which is a pair [<anchor class> <anchor relation>], where the <anchor class> is always a class name. The definitional anchor of (2.14) is [PLACE locationof] and that of (2.15) is [HUMAN compatiblewith] . We shall refer to the constraints themselves by the phrases CC[PLACE locationof] and CC[HUMAN compatiblewith].

The use of definitional anchors and the assumptions in section 2.5.4 on invocation of CC's, will enable us to write constraints as set construction expressions, as discussed in section 2.5.5.

2.5.4. The Use of Invocation Anchors

We shall assume that a CC[X r] for a class X and a relation r will be invoked only in the context of evaluating

or checking the truth value of an ($@_X$ r y), where $@_X$ is an instance of X. We shall call [$@_X$ r] the invocation anchor of CC[X r], and $@_X$ itself, the anchor. The invocation of CC[X r] may thus occur under two conditions:

(a) When executing (IR r y) or (IRN r y) (or (IR r) or (IRN r)), and MACC has an $@_X$, and the truth value of ($@_X$ r y) (or the value of ($@_X$ r)) is unknown.

(b) When executing (IR r_1 z) or (IRN r_1 z) for some z and r_1, and MACC has an $@_Y$ such that ($@_Y$ r_1 z) is dimensionally consistent. In this case CC[X r] may be invoked at an [$@_X$ r]$_\ell$ if ($@_X$ r) depends on ($@_Y$ r_1). Thus, assigning z to ($@_Y$ r_1) might affect the value of ($@_X$ r). Therefore, CC[X r] should be checked at $@_X$ under the hypothesis ($@_Y$ r_1 z).

In view of this invocation protocol we shall use in every CC a distinguished free variable called, @, which will always get bound to the anchor of the model space at the time of invocation of the CC.

2.5.5. The Use of Set Constructs

The focus of attention during the evaluation of a CC[X r] at an anchor $@_X$, is the set (s | ($@_X$ r s)). To realize the objective [Ob2] we shall seek constraints of the form

$(SP\ @_X\ s)$, in which $@_X$ and s occur free, and

$$(@_X\ r\ s) \Longleftrightarrow (SP\ @_X\ s) \quad .. \qquad (2.16)$$

SP is called the SET predicate, since one may write, for a description schema (X r Y),

$$CC[X\ r] = ((Y\ s) \mid (SP@s)) \quad ... \qquad (2.17)$$

The set expression in (2.17) may be read as "the collection of all instances, s of Y, such that (SP @ s) is true." If Y is a node, then the ASSIMILATOR will expect (SP @ s) to return a unique singleton collection, (s). On the other hand, if the description schema is (X r Z), where (Z elements Y) is true, then the ASSIMILATOR will anticipate one or more members, s, in the collection. One may also, of course, put constraints on the maximum and minimum number of candidates that (SP @ s) may return. We shall call s the set variable of the CC.

As we shall see in the ensuing sections, the ability to specify constraints in the for (2.17) with interpretation (2.16), and the conventions we have adopted on the invocations of CC's, together will make it possible for us to realize the objectives $[ob_1]$ through [ob5]. There is, however, a minor difficulty to be overcome: It is not always possible to find constraints of the form (2.16). Often one may have only a (Q @ s) such that

$$(@\ r\ s) \Rightarrow (Q\ @\ s). \quad ... \qquad (2.18)$$

In cases like this we shall write

$$CC[X\ r] = ((Y\ s) \mid (@\ r\ s)(Q\ @\ s)) \quad ... \qquad (2.19)$$

Constraints of this form are given special interpretations in the ASSIMILATOR. While evaluating (2.19) the system would expect a candidate, s, to be supplied. If no candidate is supplied then, s = ? and (@ r s) = ? is assumed, and (Q @ ?) is evaluated. This may result in the identification of a collection of candidates $(y_1\ y_2 \ldots y_k) = (y \mid (SP @ y) = ?)$, for one or more of which (@ r y) may be true. Since (@ r s) = ?, in this case the set predicate itself will evaluate to ?, if (Q @ y) ≠ NIL.

CC's of the form (2.19) are the <u>declarative CC's</u> and those of the form (2.17) are the <u>imperative</u> CC's. Using these conventions we may now rewrite (2.14) and (2.15) as shown in (2.20) and (2.21). These expressions are typical of the <u>declarative sentences</u>* in L_T :

```
CC[PLACE locationof] =
  [(HUMAN/VEGETABLE/ANIMAL/VEHICLE s)|
   ((@ locationof s)(s heldby NIL) V
    (s heldby:location @))
   ((y) (@ locationof y) ⇒ (SAFE @ y)]  ...        (2.20)

CC[HUMAN Compatiblewith] =
  [(y | (@ type:typeof y) V
       ~(y instanceof HUMAN)]                ...    (2.21)
```

In (2.20) the phrase (s heldby NIL) is a functional, interpreted as ((∀z) ~(s heldby z)). The phrase "(@ locationof s)" indicates that an s may be declared to the system. If (s heldby NIL) is true then the proper s is specified by the

---

* All CC's are constructed from <u>declarative sentences</u> in L_T. However, not all CC's are <u>declarative CC's</u>.

predicate, functional, (s heldby:location @). Notice that (2.20) is more compact than (2.14) and is oriented more towards evaluation at a given anchor, @, or given pair [@ s]. The constraint (2.21) illustrates a case where it may be more economical to store in the model space (s|~(@ r s)) than (s|(@ r s)).

In the following sections we shall discuss the interpretations given to the above CC's in the modelling context. We shall see how the objectives [ob_1] through [ob5] may be realized. We shall also present examples of <u>commonsense reasoning</u> that is used to supply reasons for the truth values in the model space for the various relations.

2.5.6. <u>Uses of CC[X r] as a function and a predicate: Examples of Commonsense reasoning</u>.

One may have two kinds of invocations of a CC[X r]:

(a) CC[X r]($@_X$) and
(b) CC[X r]($@_X$ s_o).

In both cases CC[X r] is used as a function with lambda variables @ and s, and an attempt is made to compute (s | ($@_X$ r s)). In the first case ($@_X$ r) = ? is initially assumed, and one of the following may result:

(i) (s | ($@_X$ r s)): This may happen if CC[X r]($@_X$) is imperative.

(ii) (s ($@_X$ r s) = ?): This will be interpreted as a collection of candidates for ($@_X$ r).

(iii) ? : In this case one may also get a predicate expression characterizing (s | ($@_X$ r s)) for the given $@_X$

(iv) NIL.

In case (b) also the same four possibilities exist for the result. But in cases (i) and (ii) of the result the returned collections may include s_o, thereby indicating the truth value of ($@_X$ r s_o). In both these two invocations, the set predicate of CC[X r] may be used to explain why ($@_X$ r s_o) = T, ? or NIL for a given s_o, or why ($@_X$ r) = (s | (SP $@_X$ s)). Let us consider a few examples.

Let us assume the model space for the M&C problem. Let MISSIONARY and CANNIBAL be instances of HTYPE, types of HUMANS. Let the model space have

(MISSIONARY typeof (m_1 m_2 m_3))
(CANNIBAL typeof (c_1 c_2 c_3)
(VEHICLE instance BOAT)
(HUMAN instance (m_1 m_2 m_3 c_1 c_2 c_3)).

For a missionary, m_1, then

CC[HUMAN compatiblewith] (m_1) = (m_1 m_2 m_3 BOAT).. (2.22)

as per (2.21). The reason for this will be

[(m_1 type:typeof y) V ~(y instanceof HUMAN)] (2.23)

where y is the set variable. The expression in (2.23) consists of the true literals of (2.21) for one or more items in (m_1 m_2 m_3 BOAT). In this case, there are no literals that are false or ? for all the elements in (m_1 m_2 m_3 BOAT). The reason for (m_1 compatiblewith m_2) will be

(m_1 type:typeof m_2) ... (2.24)

In this case the second disjunct of (2.23) becomes false and thus does not appear as part of the reason. We shall call expressions like (2.23) and (2.24) True Residues: (2.23) is the true residue of CC[HUMAN compatiblewith] (m_1), and (2.24) is the true residue of CC[HUMAN compatiblewith] (m_1 m_2). For a definitional anchor [X r] we shall denote its true residues by phrases of the form:

TR(CC[X r](@)) or TR(CC[X r](@ s)).

A true residue will exist for a CC, for given bindings of @ and the set variable, s, only if the set predicate of the CC evaluates to T. The true residue will consist of the sub-expressions of the parent expression, that remain after deleting all those that evaluated to NIL or ?. In cases where the set variable ranges over a collection, we shall delete only those sub-expressions that evaluated to NIL or ? for all possible bindings of s. We shall generally write residue expressions indicating explicitly the bindings of the variables. Thus, for (2.23) and (2.24) we shall write:

$$[((\text{HUMAN } @) = m_1)$$
$$((\text{HUMAN/VEHICLE } y) = (m_1\ m_2\ m_3\ \text{BOAT}))$$
$$[(@\ \text{type:typeof } y)\ V\ \sim(y\ \text{instance of BOAT})]]\ldots \quad (2.23a)$$

$$[((\text{HUMAN } @) = m_1)((\text{HUMAN } y) = m_2)(@\ \text{type:typeof } y)] \quad (2.24a)$$

For a CANNIBAL, c_1, the reason for (m_1 compatiblewith c_1) = NIL will be

$$[\sim(m_1\ \text{type:typeof } c_1)(c_1\ \text{instanceof HUMAN})] \quad (2.25)$$

which is obtained by taking the negatinn of the <u>False Residue</u> of CC[HUMAN compatiblewith] (m_1 c_1). In this case, of course, CC[HUMAN compatiblewith](m_1 c_1) = NIL. The False Residue will consist of the sub-expressions that remain after deleting all those that evaluated to T or ?. Of course, the parent expression itself should evaluate to NIL. We shall use phrases of the form FR(CC[X r](@)) and FR(CC[X r](@ s)) to denote false residues of CC's.

One may similarly define also <u>Unknown Residues</u>, UR(CC[X r](@)) and UR(CC[X r](@ s)). These will exist only when the CC evaluates to ? and will be obtained by deleting all the sub-expressions that evaluate to NIL or T in the set predicate, for given bindings of @ and s. In the M&C problem, suppose there were more that three MISSIONARIES. In this case the model space will contain the functional

$$(\text{MISSIONARY typeof } (m_1\ m_2\ m_3\ ?)), \quad (2.26)$$

where the ? in the collection indicates that there may be more. In MDS, a new element may be added to a collection only if the collection contained ?. Thus, one could make it impossible to have more than two HTYPEs, by setting in the model space:

(HTYPE instance (MISSIONARY CANNIBAL)).

In the case (2.26), (2.21) will evaluate to T for @ = m_1 and y = (m_1 m_2 m_3 BOAT). However, for y = ? both literals in (2.21) will evaluate to ?, producing the unknown residue:

[[(@ = m_1)(y = ?) (@ type:typeof y) V ~(y instanceof HUMAN)] (2.27)

This unknown residue may be viewed as characterizing (x | (m_1 compatiblewith x)). In this case the residue expression happens to be identical to the set predicate. But in general, the residue expressions will be subexpressions of the set predicate. The residue extraction process is a part of the commonsense reasoning process. It is defined for both propositional and quantified expressions in chapter 3. The relationship between residues and reasons is summarized below in Table II.

TABLE II: Residues and Reasons

TRUTH VALUE of CC[X r](@) or CC[X r](@ s)	Reason for CC[X r](@) or CC[X r](@ s). The set variable is optional below.
T	(TR(CC[X r](@ --)
?	(UR(CC[X r](@ --)
NIL	~(FR(CC[X r](@ --)

In the problem solving process the residues (reasons) are used as the basis for learning and domain specific specialization. Let us now consider a few more examples.

Table III shows all the CC's used in our domain, and Table IV shows the definitions associated with SAFE. The command (QSCC: <CC-exp> <definitional anchor>) is used to define CC's in the current implementation of MDS. This command is part of the subsystem called QUEST, which is used for defining the CC's and transformations in a domain. Predicates like SAFE are called CC-macros. They are invoked as macros within CC's and transformations. Each CCMACRO has a name, declared arguments, the macro expression and a context. Thus, a COMACRO like SAFE may have different definitions in different contexts. The definitions of SAFE in M&S and F&S contexts are shown in Table IV. The command QSCCM: is used to define CCMACRO's in MDS.

Let us now consider some of the possible residues associated with CC[VEHICLE holding]. This CC is shown in Table III, and is reproduced below, for convenience:

CC[VEHICLE holding]:
[(HUMAN/VEGETABLE/ANIMAL x) |
(@ holding x)(@ holding: # : ≤: capacityof @)
((y)(@ holding y) ⇒ (SAFE x y)] ... (2.28)

TABLE III: Consistency Conditions of the Transportation Domain.

CC1: CC[HUMAN compatiblewith].
[(HUMAN/ANIMAL/VEGETABLE/VEHICLE s) |
(@ type:typeof s) V ~(s instanceof HUMAN)].

CC2: CC[VEHICLE holding]
[(HUMAN/ANIMAL/VEGETABLE s) |
(@ holding s)(@ holding:# : ≤ :capacityof @)
((y)(@ holding y) ⇒ (SAFE s y))]

CC3: CC[ANIMAL compatiblewith]
[(HUMAN/VEGETABLE/ANIMAL/VEHICLE s) |
(@ type:typeof s) V (@ instanceof HUMAN) V
~(@ type HERBIVORE)(s instanceof VEGETABLE)]

CC4: CC[VEGETABLE compatiblewith]
[(s | (@ compatiblewith s) V
(s instanceof VEGETABLE)]

CC5: CC[PLACE locationof].
[(s | ((@ locationof s)(s heldby NIL) V
(s heldby:location @))((y)(@ locationof y)
⇒ (SAFE s y)]

CC6: CC[ITEMS heldby]
(s | (@ heldby s) ~(s elementof @)].

The CC has the form (2.19) and is thus a declarative CC: It may be used to check a given (@ holding s_o) but cannot be used to find (s | (@ holding s)). In the second conjunct of (2.28) the relation name "#" occurs in the path "holding:#:≤: capacityof". This is used to get the cardinality of (@ holding). In the context of #, collections are interpreted as, sets:

Table IV: COMACROS in the domain.

```
[OSCCM:   SAFE (X Y) "M&C"
[(X location:locationof Y) ⇒
 ((SOME ITEMS s)(X location:locationof s)
 (((#OF MISSIONARY s) ≥ (#OF CANNIBAL s)) V
  ((#OF MISSIONARY s) is 0)]].
```

Note: The third argument of QSCCM: is the context.

```
[QSCCM:   SAFE (X Y) "F&S"
[(X compatiblewith Y) V
 (X location:locationof Y) ⇒
 (((SOME HUMAN h)(X location:locationof h)]].
```

$(\{x_1\ x_2 \ldots x_n\}\#) = n$, and $(\{x_1\ x_2 \ldots x_n\ ?\}\#) \geq n$ but still the relation $(\{x_1\ x_2 \ldots x_n\ ?\}\#n)$ has the truth value ?. Thus, in a comparison like $(\{x_1\ x_2\ ?\}\#:\leq 2)$ its truth value will be ?. So also, $(\{x_1\ x_2\ ?\}\#:\geq 4)$ will be ?. But, $(\{x_1\ x_2\ ?\}\#:\geq 2)$ will have truth value T.

Let us assume that initially (BOAT holding ?) is true in the model space. In this case CC[VEHICLE holding](BOAT) will evaluate to ? with the unknown residue:

```
[(BOAT holding x)( BOAT holding: #:≤:capacityof BOAT)
 ((y)(BOAT holding y) ⇒(SAFE x y)] ...                (2.29)
```

In the above expression we may ignore the literal "(BOAT holding x)". since it is part of the declarative nature of the CC: For every assertion (BOAT holding x) will be either true or false by hypothesis. Let us now assert (BOAT holding x_o). This will cause the model space entry (BOAT holding $(x_o\ ?)$). The evaluation of CC[VEHICLE holding](BOAT x_o)

will evaluate to ?, because both (BOAT holding:#:≤:capatityof BOAT) and ((y)(BOAT holding y) ⇒ (SAFE x_o y))* will evaluate to ?, leading to the unknown residue:

[(BOAT holding:#:≤: capacityof BOAT)
((y)(BOAT holding y)⇒ (SAFE x_o y))] ... (2.30)

Notice that the set variable x in (2.29) has been replaced by x_o in (2.30). Thus, all future additions to the BOAT should be SAFE with x_o.

Let us now suppose that (BOAT capacity 4) is true, and when (BOAT holding x_o) was asserted (BOAT holding (x_1 x_2 ?)) was in the model space. In this case, the unknown residue will be the same as (2.29) for the collection (x_o x_1 x_2 ?). We have assumed that the SAFE predicate is not contradicted for x_o.

If the SAFE predicate was contradicted then for some element x, in (x_1 x_2), (SAFE x_o x) would have been NIL. In this case the model space would remain unchanged, and the following residue would have been supplied for not accepting (BOAT holding x_o):

[(VEHICLE @) = BOAT)
(HUMAN/ANIMAL/VEGETABLE y) = (x_o x_1 x_2 ?))
(SAFE x_o y)] ... (2.31)

The reason would be,

---

* In this case (SAFE x_o x_o)=T and (SAFE x_o ?)=?. The bound variable y acquires the binding ? because (BOAT holding (x_o ?)) exists in the model space.

$$[((\text{VEHICLE } @) = \text{BOAT})$$
$$((\text{SOME/HUMAN/ANIMAL/VEGETABLE } y) = (x_o\ x_1\ x_2\ ?))$$
$$(\sim(\text{SAFE } x_o\ y))] \qquad (2.32)$$

which is the negation of (2.31). In a problem solving context, reasons like this may be made use of to avoid repeating same kind of mistakes. Also, reasons explicating true residues may be made use of to make the right choices based on past experience. The properties of residues (and reasons) that make them useful in a problem solving context are discussed in chapter III.

The reasons obtained from the CC's at a given anchor are not sufficient to explain or guide an updating process. The CC itself may supply only the necessary conditions. To consider the complete updating process it is necessary also to analyze the way relations interact in the model space. This is discussed in the next section, where an example of commonsense reasoning in the context of relation interactions is presented. Again we shall see that the form and interpretations of CC's play an important role in identifying and controlling the interactions.

2.5.7. Interaction Between Relations: Their Recognition and Control

2.5.7.1 DONLISTS and DETLISTS

Definition 1: Depends on

([Q_X r] dependson [Q_Y t]) if there exist a z (z could be ? or NIL) such that (Q_Y t z) or (z t' Q_Y)(or ~(Q_Y t z) or ~(z t' Q_Y)) occurs in a true, false or unknown residue of CC[X r](Q_X) or CC[X r](Q_X s_o) for some s_o.

In this case we shall say that [Y t] is an element of DONLIST [X r].

Definition 2 : Determines

([Q_Y t] determines [Q_X r]) if ([Q_X r] dependson [Q_Y t]). In this case, we shall say that [X r] is an element of DETLIST [Y t].

Notice that [X r] ϵ DETLIST [Y t] does not necessarily mean that for any given Q_Y and Q_X ([Q_X r] dependson [Q_Y t]). It only implies that there exist Q_X and Q_Y such that ([Q_X r] dependson [Q_Y t]). We have the following formulas:

(([Q_X r] dependson [Q_Y t]) $\Leftrightarrow$ ([Q_Y t] determines [Q_X r]))
(([X r] elementof DETLIST[Y t]) $\Leftrightarrow$ ([Y t] elementof DONLIST[X r]).
(([Y t] elementof DONLIST[X r]) $\Leftrightarrow$ ((SOME X Q_X)(SOME Y Q_Y)
([Q_X r] dependson [Q_y t])))
(([X r] elementof DETLIST[Y t]) $\Leftrightarrow$
((SOME X Q_X)(SOME Y Q_Y)([Q_Y t] determines [Q_X r])))

The DONLISTs and DETLISTs for the definition anchors may be obtained by analyzing the forms of CC's in a domain. These may be used in a variety of ways to identify, anticipate, control and respond to situations that arise in updating processes. We shall present below an example of the kind of analysis that may

be done to construct DETLISTs.

We shall also introduce the concept of definitional filters that are used to direct search for all $\mathcal{C}_X$ such that $[\mathcal{C}_X\ r]$ depends on $[\mathcal{C}_Y\ t]$ for a given $\mathcal{C}_Y$. We shall discuss ways of using filters to minimize search and checking during updating processes.

2.5.7.2: The Dimensionality of a CC and its Dependency Graph

Let us consider again CC[PLACE locationof] shown in (2.20). One may construct for this CC a, so called, dependency graph as shown in figure 4. The arcs in this graph represent the relation names that appear in the CC, with the associated negation signs, if any. The nodes represent class names that are used in the CC either explicitly or implicitly.

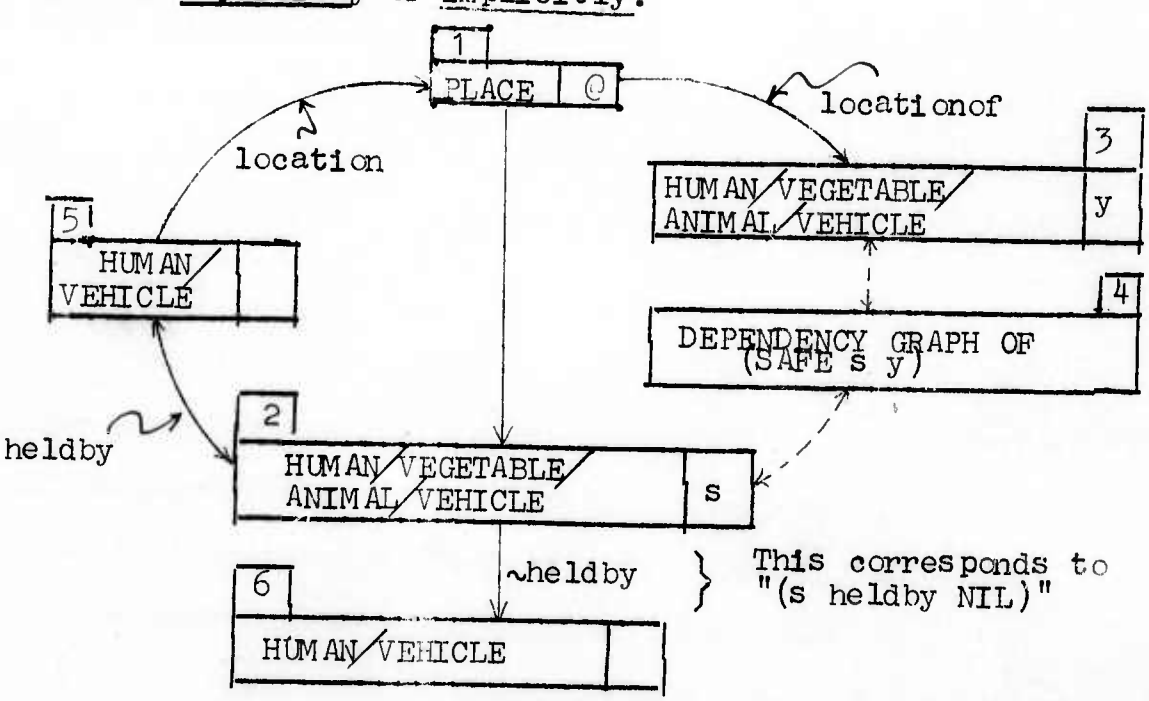

Fig 4: Dependency Graph of CC PLACE locationof.

For example, the expression

"((∀y)(@ locationof y) ⇒ (SAFE s y))"

that occurs in (2.20) uses the class disjunction (HUMAN/VEGETABLE/ANIMAL/VEHICLE y) explicitly, because there exists a bound variable y, that represents the instances of these classes that are used in an evaluation of the CC. This expression is represented in figure 4 by nodes 1, 2, 3 and 4, and the arcs "locationof" and "<->", where "<->" represents connections to the dependency graph of (SAFE s y).

In the case of the expression

"(s heldby:location @)"

which also occurs in (2.2) the (HUMAN\VEHICLE y) such that ((s heldby y) ⇒ (y location @)) is said to be implicitly used in the CC. The CC has no bound variable corresponding to the y above. Thus, node 5 appears in figure 4 without an associated variable.

The classnames used implicitly in a CC may be determined from the relation paths used in the CC and the description structures defined for the domain. We shall call the analysis used to identify the implicit and explicit class names in a CC, the dimensional analysis. For a functional like "(s heldby:location @)" we shall represent its dimension as

[((HUMAN/VEGETABLE/ANIMAL/VEHICLE s) heldby)
((HUMAN/VEHICLE) location) (PLACE O)] ... (2.35)

This dimension is consistent with the description schemas defined for the domain. A CC itself is said to be dimensionally consistent if all its literals and functionals are dimensionally consistent. The analysis of dimensional consistency may also be used to find missing relation names in relation paths, missing ranges of variables, and also errors in a CC. The dimensional consistency checking subsystem of the current implementation of MDS was written by Joel Irwin.

The dependency graph of a CC portrays its dimensionality. It may be used to construct DONLISTs, DETLISTs and definitional filters. This is discussed in the next section.

2.5.7.3. Construction of DONLISTs & DETLISTs

Let us for a moment ignore the implications of the "(SAFE s y)" predicate in figure 4. The general rules for constructing DON and DET lists from a dependency graph is given below:

DONLIST - RULE

[Y t] ϵ DONLIST[X r] if the class Y occurs in a node in in the dependency graph of CC[X r], and the arc with label t or $\sim$t(t' or $\sim$t') emanates (impinges) on the node Y in the dependency graph CC[X r]. t' is the inverse of t.

In this case [X r] ∈ DETLIST[Y t].

From figure 4 one thus obtains that there may exist a PLACE, p, a HUMAN/VEHICLE, h, a HUMAN/VEGETABLE/ANIMAL/VEHICLE x, etc. such that

```
([p locationof] dependson
     ([h holding ][p locationof]
      [x location]  etc. ...  ))
```
(2.36)

The symbol h in (2.36) denotes the implicit use of HUMAN/VEHICLE in "(s heldby:location @)". Expanding the disjunction of classes over which the variables in (2.36) range, we get by the DONLIST rule that

```
DONLIST[PLACE locationof] =
     ([HUMAN holding][VEHICLE holding]
      [HUMAN location][VEHICLE location]
      [PLACE location of] etc.)
```
(2.37)

Thus we get

```
[PLACE locationof] ∈ DETLIST[HUMAN holding]
                     DETLIST[HUMAN location]
                     DETLIST[VEHICLE holding]
                     etc.
```
(2.38)

This implies that, when (h holding x) is asserted (by using an IR Command, say) for a HUMAN, h, then CC[PLACE locationof] should be checked for all places p such that ([h holding] determines [p location of]). During this checking one may, for example, discover that (h location) = p and (x location) = g are not the same. Thus (h holding x) cannot be accepted by

the model space without violating CC[PLACE locationof]. The reason for this violation would be:

[(x heldby:location p)~(p location of x) V
~(x heldby:locationof g) ~(x heldby NIL)
(g locationof x)] ... (2.39)

This reason is true under the hypothesis (h holding x). We shall discuss in the next section the commonsense reasoning process that may be used to identify treasons like (2.39)

We shall also consider efficient ways for directly identifying for a given [h holding] all the PLACEs, p, such that ([p locationof] depends on [h holding]).

2.5.7.4: Definitional Filters

For a given [X r] ∈ DETLIST[Y t] and given Q_Y our problem is to find

$(Q_X | ([@_X\ r]\ \text{dependson}\ [Q_Y\ t]))$.

This search may, of course, be confined to only the instances of X. Still it may be large search. In practice it is necessary to have a better control over this search, and where feasible eliminate the search altogether.

The problem of devising schemes to efficiently control and direct this search is the so called "Frame Problem". The solution of this problem is not only domain dependent, but

within a domain it is dependent on the particular collections of objects that exist in the model space. It is not thus possible to specify a general solution to the "frame problem". One may only specify schemes where in the problem may be kept under control, and unforeseen combinatorial explosions are avoided. The forms of CC and their invocation control make it possible to state general schemes to keep the "frame problem" under check. What is more, one may have MDS itself compile the necessary control structures from an analysis of the CC's of the domain.

We shall discuss two ways of controlling search in a frame interaction:

(a) One is by the Definitional Filters which are compiled by MDS, and

(b) the other is by the use of representational facilities in the model space which a user may use to create appropriate representations for a domain.

Part (a) is discussed below and part (b) is discussed in section 2.5.9. Normally, when an (x r y) is asserted, in order to accept it, it may be necessary to make certain secondary changes in the model space. This may, in general result in a chain of updating processes. It is essential to have facilities for controlling this process. Devices called Focus Lists are used in MDS to control these. These are discussed in section 2.5.8.

Definitional Filters

A definitional filter DF[X r][Y t] is used to compute a set $S_X^r[@_Y\ t]$ such that

$$\{@_X\} \supset S_X^r[@_Y\ t] \supset \{@_X | ([@_X\ r]\ \text{dependson}[@_Y\ t])\}. \quad (2.39)$$

where $\{@_X\}$ is the set of all instances of X. A DF filters out of $\{@_X\}$ all x such that [x r] does not depend on $[@_Y\ t]$. The DF may be stated as

$$DF[X\ r][Y\ t] = ((X\ x)|DFP\ (@_Y\ s\ x)) \quad (2.40)$$

where $@_Y$, s and x are the free variables of the Definitional Filter Predicate, DFP. The DF is anchored at [Y t]. Thus, $@_Y$ is the anchor variable of the DF; x is its set variable; and s is the, so called, change variable: $(@_Y\ t\ s)$ is the change that is being attempted at $[@_Y\ t]$. Using such a DF, for a given $@_Y$ and s, one may compute all the affected $@_X$. The dependency graph of CC[X r] may be used to construct DF[X r][Y t], if [X r] is in DETLIST[Y t]. The construction of these filters is based on the following observation.

Let us assume that dependency graphs are always connected graphs. Consider the graph in figure 4. If [@ locationof] depends on some $[@_Y\ t]$ for a Y and t in the graph, then there will be a path w, from @ to $@_Y$ in the graph. @ itself may, therefore, be reached from $@_Y$ via the inverse path w'. Suppose, $@_Y$ = h for a HUMAN, h, and @ is a PLACE, p. Let [p locationof]

depend on [h holding]. The paths in figure 4 that might lead to h from p are (again we ignore (SAFE s y)):

[(p locationof h) V (p locationof s) ~(s heldby h) V (p locationof s)(s heldby h)]

Thus p itself should be reachable from h via one of the paths in the disjunction

[(h location p) V ~(h holding s)(s location p) V (h holding s)(s location p)] ... (2.41)

Thus, at a given h, if (h holding s) was the change at (h holding) then the places affected by it will be a subset of

(p|(h location p) V ~(h holding s)(s location p) V (h holding s)(s location p)) ... (2.42)

We may now write

DF[PLACE locationof][HUMAN holding] =

[(PLACE p)|(@ location p) V ~(@ holding s)(s location p) V(@ holding s)(s location p)] ... (2.43)

where @ is a HUMAN, and s is the change variable. This DF will be anchored at [HUMAN holding]. Thus, if (h holding x) was newly asserted, then the system would evaluate the above DF for (@=h) and (s=x), and find that it had to check CC[PLACE locationof] at the places p = (h location) and g = (s location), under the hypothesis (h holding x). At both these places, p and g, a

contradiction will be encountered for the following reasons.

We have the assumptions (h holding x), (h location) = p and (x location) = g. For the discussion below, we shall also assume that the SAFE predicate is true, and altogether ignore this predicate in the residues and reasons. At the PLACE, p,*

CC[PLACE locationof] (p x) = T and
TR(CC[PLACE locationof] (p x)) = (x heldby:location p)

By (2.16) it, therefore, follows that (p locationof x) should be true in the model space. This contradicts the assumption, (g locationof x). The reason for this contradiction will be the negation of the false residue of

(p locationof x) $\Leftrightarrow$ CC[PLACE locationof](p x).

The false residue of a form (u $\Leftrightarrow$ v) is the same as the false residue of the form (uv V $\sim u \sim v$). In the above case u is false and v is true. We have the following residue equation.**

FR(uv V $\sim u \sim v$) = FR(uv) V FR($\sim u \sim v$) = FR(u) V FR($\sim v$)

since u is false and v is true. However, FR($\sim v$) = $\sim$TR(v). Thus, we get, that the reason for the contradiction is

---

* Please check with CC[PLACE locationof] shown in (2.20).

** These are discussed in chapter III.

[A](~FR(u)^TR(v)) = ~(p locationof x)(x heldby:location p)

In the case of the PLACE, g, CC[PLACE locationof](g) will evaluate to NIL and false residue will be:

FR(CC[PLACE locationof](g x)) =
[(x heldby NIL) V (x heldby:location g)].

and (g locationof x) is true. Thus, in this case the reason for the contradiction will be

[B]. [(g locationof x) ~(x heldby:locationof g)
~(x heldby NIL)].

The resultant reason will be the disjunction of [A] and [B], namely the expression in (2.39). To accept the assertion (h holding x) this reason should now be made to disappear. That is, it should be made to evaluate to NIL or ? in the model space. In general, it is necessary to use a "means-end analysis" scheme to realize this objective. We shall present in the next section a simple way of doing this, that works for most of the cases arising in the model space. Devices called, Focus Lists (FL's) are used in the model space for this purpose. We shall conclude this section with a summary of the properties and conventions associated with interaction checks (frame checks) in the model space.

Frame Checking in the Model Space

The DF's are used to identify interactions that are not

directly implied by the description structures in a domain. The interactions directly implied by the description structures in a domain are illustrated by the following example.

Suppose (h holding x) was newly asserted, for an ANIMAL, x, and at the time this assertion was made (m holding x) was true in the model space. In our domain we have the schema

[S8] (ITEMS heldby HUMAN\VEHICLE)

and thus only one HUMAN may hold an object.* Therefore, the assimilator will postulate automatically all the following relations:

[(h holding x) (x heldby h) ~(m holding x) ~(x heldby m)]

... (2.44)

The assertions for "holding" and "heldby" appear together in (2.44) because of the assumption (2.4), which is a part of the built in structure of the model space. The negated assertions appear because of the schema [S8]. This knowledge will be a part of the domain dependent processors compiled for the domain. To check for the consistency of (2.44) the following CC's should be checked:

CC[ANIMAL heldby](x) (2.45a)
CC[HUMAN holding](h) (2.45b)
CC[HUMAN holding](m), (2.45c)
CC[X r]($@_x$), (2.45d)

---

*It is also true that while a VEHICLE is holding something nothing else may hold the same object. A description scheme where more than one person or vehicle may hold an object is presented in Appendix I.

for every [X r] such that $\mathcal{Q}_X$ is a member of one or more of the following sets:

DF[X r][ANIMAL heldby](x h)
DF[X r][ANIMAL h ldby](x m)
DF[X r][HUMAN holding](m x)
DF[X r][HUMAN holding](h x) ... (2.46)

For each DF above the argument pair is (<anchor><change>). The assertions in (2.44) may be accepted only if none of the CC's above produce a contradiction.

In our domain the CC's (2.45a) through (2.45c) do not exist. Where a CC does not exist for an anchor [X r] we shall assume that the CC is $(y|(\mathcal{Q}_x\ r\ y))$, i.e. any declared y is acceptable. Thus, the only checks in the case of (h holding x) will be those resulting from the interactions, namely those implied by (2.45d) and the DF's in (2.46).

In general, for any $(\mathcal{Q}_X\ r\ y)$ the interaction between r and its inverse, and r and itself, is part of the structural knowledge built into a domain. We shall therefore ignore all definitional filters of the forms:

DF[X r][X r] and DF[X r][Y r']

for all Y for which (X r Y) is true -- i.e. the scheme (X r Y) exists. We thus have the following lemma characterizing the conditions for the existence of DF's:

LEMMA 2.1:*

$$[(\text{EXIST DF}[X\ r][Y\ t]) \Longleftrightarrow (\text{EXISTS CC}[X\ r]'$$
$$([Y\ t]\ \text{elementof DONLIST}[X\ r])$$
$$\sim([Y\ t] = [X\ r])\ (\sim(X\ r\ Y)\ V\ \sim(t\ \text{inverseof}\ r))]$$

Frame Filters

Besides definitional filters, DF[X r][Y t], one may also have in MDS the so called frame filters (FF's), FF[X r][Y t]. An FF[X r][Y t] may exist only if DF[X r][Y t] exists, and if the FF exists then the subset

$$S_X^r[\mathcal{C}_Y\ t] = (DF[X\ r][Y\ t](\mathcal{C}_Y\ s) \cap FF[X\ r][Y\ t](\mathcal{C}_Y\ s)) \tag{2.49}$$

Frame filters may be problem dependent and may be assigned to an anchor during a problem solving process. It may also be defined at the time of domain definition. Examples of use of frame filters are not discussed in this paper.

The subsystem for building dependency graphs and definitional filters was built by John Ng, in the current implementation of MDS.

* The Unary predicate EXISTS is used here with the obvious connotation.

2.5.8. Focus Lists and the updating Process

Whereas DF's and FF's are used to identify and select primary interactions, one may think of Focus Lists as controlling secondary changes, induced by an assertion (x r y). With each [X r] one may associate two kinds of focus lists: The positive focus list, PFL[X r], and the negative focus list, NFL[X r]. From among all [Q_Y t] such that ([Q_X r] dependson [Q_Y t]), the NFL[X r] is used to select those that should remain unchanged in the updating process. Thus, NFL[X r] characterizes the stable relations on which [Q_X r] may dependon. If there is an inconsistency at [Q_X r] then none of the stable relations may be changed in order to resolve the inconsistency. Similarly, PFL[X r] characterizes all the unstable relations on which [Q_X r] may depend on. Thus, to resolve an inconsistency at an [Q_X r], one or more of the unstable relations may be changed.

An element of PFL[X r] is of the form PFL[Y t][X r]. So also, an element of NFL[X r] is of the form NFL[Y t][X r]. In both cases, [Y t], should belong to DONLIST[X r]. Each PFL[X r] (NFL[X r]) is anchored at [X r]. Notice the duality between DF's and FL's (Focus Lists): A DF is of the form DF[X r][Y t] where [X r] ϵ DETLIST[Y t], where as an FL is of the form FL[Y t][X r].

Each element of an FL is itself again a set construction expression of the form.

$$\text{PFL}[Y\ t][X\ r] = ([y\ z] \mid ([Q_X\ r]\ \text{dependson}\ [y\ t])\ (y\ t\ z)(\text{PFLP}\ Q_X\ y\ z))$$

$$\text{NFL}[Y\ t][X\ r] = ([y\ z] \mid ([Q_X\ r]\ \text{dependson}\ [y\ t])\ (y\ t\ z)(\text{NFLP}\ Q_X\ y\ z))$$

Clearly,

$$\text{PFL}[Y\ t][X\ r](Q_X) \cap \text{NFL}[Y\ t][X\ r](Q_X) = \text{NIL} \qquad (2.50)$$

We shall usually assume the conjuncts "([Q_X r] dependson[y t]) (y t z)" and simply write PFL and NFL expressions as

$$\text{PFL}[Y\ t][X\ r] = ([\ y\ z] \mid (\text{PFLP}\ Q_X\ y\ z)) \qquad (2.51)$$

$$\text{NFL}[Y\ t][X\ r] = ([\ y\ z] \mid (\text{NFLP}\ Q_X\ y\ z)) \qquad (2.52)$$

Also, when PFLP or NFLP is vaccuous -- i.e. always T -- then we shall simply say that [Y t] itself is a member of PFL[X r] or NFL[X r]. Thus, if [Y t] is a member of NFL[X r] then for an [Q_X r] all (Q_Y t) such that ([Q_X r] dependson [Q_Y t]) are stable. So also, if [Y t] is a member of PFL[X r] then all (Q_Y t) such that ([Q_X r] dependson [Q_Y t]) are unstable.

In our domain we may have for example,

$$\text{NFL}[\text{PLACE locationof}] = ([\text{HUMAN holding}][\text{VEHICLE holding}]). \qquad (2.53)$$

indicating that when (Q_{PLACE} locationof) changes then the pertinent "holding" and "heldby" relations should remain stable. Also, we may have

PFL[PLACE locationof] =
([HUMAN location][VEGETABLE location]
[ANIMAL location][VEHICLE location]) (2.54)

Thus, if there is an inconsistency at an (Q_{PLACE} locationof) then one may attempt to resolve it by changing the location of a HUMAN, ANIMAL, VEGETABLE or VEHICLE. Let us consider an example.

Suppose a HUMAN, h, is holding an ANIMAL, x, and h changes location from p to g. Let us assume that initially the following is true:

[(h location p)(x location p)(p locationof(h x))
(h holding x)(x heldby h)] (2.55)

To move the HUMAN from p to g the following should be made true in the model space (this is obtained directly from the description structures involved):

[(h location g)(g locationof h)
~(h location p) ~(p locationof h)] (2.56)

To accept these the following checks should be done:

CC[PLACE locationof](p)
CC[PLACE locationof](g) (2.57)

and for every Q_x, such that Q_x is a member of one or more of the following sets,

DF[X r][PLACE locationof](p h)
DF[X r][PLACE locationof](g h)
DF[X r][HUMAN location](h p)
DF[X r][HUMAN location](h g) (2.58)

one has to check

CC[X r]($\mathcal{Q}_X$). (2.59)

If we again ignored the SAFE predicate, as we shall see below, none of the above DF's would exist. In Table III one may notice that "location" and "locationof" occur only in CC[PLACE locationof] and in CC[VEHICLE holding]. In the latter, it occurs via the SAFE predicate. If we ignored SAFE -- assuming it to be always true -- then the only source of interaction with "location" and "locationof" is CC[PLACE locationof]. Thus, in (2.58), r would be either "location" or "locationof" and X would be HUMAN/VEGETABLE/ANIMAL/VEHICLE. By lemma 1, such DF's cannot exist. Thus, in this case we have no relation interactions to check. The only CC's to check are those in (2.57). Let us suppose that

TR(CC[PLACE locationof](p h)) =
((h heldby NIL)
((∀ y)(p locationof y) ⇒ (SAFE h y)] (2.60)

and,

TR(CC[PLACE locationof](p)) =
[(Q = p)(s = (x ... ?))
((s heldby NIL)V(s heldby:location Q))
((∀ y)(Q locationof y) ⇒ (SAFE s y))] (2.61)

The residue (2.60) will evaluate to T under hypothesis (2.56), but (2.61) will evaluate to NIL, because for (s=x), and (o=p) the expression ((s heldby NIL)V(s heldby:location p)) will be NIL. This now contradicts (2.16), namely (p locationof x) $\Longleftrightarrow$ CC[PLACE locationof](p x) producing the reason

[~(x heldby NIL)(p locationof x)
~(x heldby:location p)].. (2.62)

In the case of CC[PLACE locationof](g) we have the opposite situation:

CC PLACE[locationof](g x) = T but (g locationof x) is false, because x is still at p in the model space. This will produce the reason:

[(x heldby:location g) ~(g locationof x)] (2.63)

The combined reason for failure at the definitional anchor [PLACE locationof] will be the disjunction of (2.62) and (2.63): Let (R x p g) denote this disjunction:

(R x p g) =

[~(x heldby NIL)(p locationof x)
~(x heldby:location g) V
(x heldby: location g) ~(g locationof x)] (2.64)

To eliminate this cause for failure we shall try to make (R x p g) = ?. To do this one or more relations in (2.64) should be set to ?. The values of the relations implied by NFL[PLACE locationof] cannot be changed. Thus none of the

"holding" or "heldby" relations may be changed (Please see NFL in (2.53)). So also, none of the relations in the hypothesis (2.56) may be changed. Deleting from (2.64) all the "holding" ("heldby") relations, and the relations of the hypothesis we have left

$$[(p \text{ locationof } x) \vee \sim(g \text{ locationof } x)] \qquad (2.65)$$

as the only candidates for change. Indeed, both these relation values may be changed since they both belong to PFL[PLACE locationof]. Let us set

$$[(p \text{ locationof } x) = (g \text{ locationof } x) = (x \text{ location}) = ?] \qquad (2.66)$$

in the model space. Then, under the combined hypothesis (2.56) and (2.66) the reasons for the contradiction disappear. Also, in this case, the evaluation of CC[PLACE locationof](g) will automatically set (x location) = g and (g locationof) = (h x..?). This will complete the process of assimilating the assertion (h location g).

In general, in the model space, we shall always attempt to eliminate the reasons for a contradiction by forcing it to evaluate to ?.

The focus list should be made more selective than the ones shown in (2.53) and (2.54). It is quite possible that not all "holding" relations should remain stable. Thus, for example,

h may be holding more than one object, some may be FIXED objects -- where FIXED is, say a type of object -- and others may be MOVABLE objects. One may then have the rule that FIXED objects cannot change locations, only MOVABLE ones can. This rule may be captured by the following FL expressions:

$$\text{NFL[HUMAN holding][PLACE locationof]} = ([@_{\text{HUMAN}}\ z] \mid \sim(z \text{ type FIXED})) \qquad (2.67)$$

Only for objects z that are not FIXED should ($@_{\text{HUMAN}}$ holding z) remain stable. Similarly,

$$\text{PFL[HUMAN holding][PLACE locationof]} = ([@_{\text{HUMAN}}\ z] \mid (z \text{ type FIXED})) \qquad (2.68)$$

For FIXED objects ($@_{\text{HUMAN}}$ holding z) may be changed. Thus, when z moves, z would let go his hold on FIXED objects and take with him only the MOVABLE ones.

Focus list conditions like this may be defined at domain definition time or at problem solving time. The DF, FF and FL mechanisms provide a practically unlimited and continuous control of frame interactions, and secondary updating. The focus lists not only provide guidance for secondary updating, but also provide a formalism to describe updating criteria. Thus, strategies learnt in an updating process may be summarized as focus lists, for future use.

The focus list mechanism will be automatically invoked in an updating process to do means-end analysis, when necessary, unless it is blocked off by the, so called, filter switch, which can be associated at some time with invocation anchors, $[O_x\ r]$. If for an [X r] there are no focus lists then it is assumed that all the relations are stable in the context of every $[O_x\ r]$. Thus, if the filter switch is on for an $[O_x\ r]$, or if there are no focus lists for an [X r], then the means-end analysis step will be skipped. The command FR, (Force Relation) is used to force the invocation of the means-end analysis processes during updating.

In the next section we shall discuss some representational shifts in the model space that would enable the system to completely avoid the search for frame interactions and secondary updating, in cases where locations of objects change. In effect in the new representations, the objects carried by a HUMAN/VEHICLE will implicitly move with the HUMAN/VEHICLE. This shift in representation is achieved by the use of anchored transformation rules and the dummy storage flag (which was discussed in section 2.3.2, item (iii)).

2.5.9. Anchored Transformation Rules

The Anchored Transformation Rule, ATR[X r] has the general form:

```
[(NIL  <NIL-action>)
 ( ?   <?-actions >)
 ( T   <T-actions >)                                    (2.69)
```

where NIL, ? and T are the possible outcomes of all the consistency and interactions check at an $[Q_X\ r]$. If the checks result in NIL then the <NIL-actions> will be executed. Hopefully, these actions will remove the cause of the contradiction. If the checks evaluate to ? then the <?-action> will be executed. These may find some or all of the unknown values in an updating process. The <T-actions> when executed may cause the "side effects" necessary in the domain when $(Q_X\ r)$ is updated. In this section we shall discuss two kinds of uses of ATR's. One is a prescription for the kind of updating that was discussed in the previous section. The other is a shift in representation that eliminates the need for secondary updates.

Each ATR may, by convention, use implicitly the following arguments:

i) Q_X : the anchor
ii) s : the change at $(Q_X\ r)$
iii) OLDVAL : Old value of $(Q_X\ r)$
iv) NEWVAL : New value of $(Q_X\ r)$, and
v) REASONS: The reasons obtained from all the checks.

Let us associate with [PLACE locationof] the following ATR:

```
ATR[PLACE locationof] =
   (NIL ((x | (s holding x) ~(x type FIXED))
              (ASSERT (x location ?)))
        ((x | (s holding x)(x type FIXED))
              (ASSERT ~(s holding x)].                    (2.70)
```

The first component of <NIL-action> asserts (x location) = ? for all x that in effect satisfy (2.67). The second component asserts ~(s holding x) for all x that satisfy (2.68). The command phrases follow the syntax:

```
<command phrase> :: = <action> | <commandphrase> ...
  ...<command phrase> | (<bindingcondition>
                          <commandphrase>)
```

The binding condition is used to bind variables which participate in the <action>. The action itself will be executed only if the <binding condition> is successful. In (2.70) the binding conditions are set expressions. In cases like this the indicated action is performed on all the elements of the set.

The above ATR gives a prescription for using the focus list predicates associated with [PLACE locationof]. This may result in avoiding some search and decision at the updating time, at the expense of flexibility. The use of ATR's to change representations in the model space is shown below:

Let us associate with [HUMAN holding] and [VEHICLE holding]

the following ATR, and modify ATR[PLACE locationof] as shown in (2.73). These ATR's and their operations are discussed below:

```
ATR[HUMAN holding] = ATR[VEHICLE holding] =
   [(T ?)
    (((SOME PLACE p)(Q location p))
     ((y | (y elementof NEWVAL) ~(y elementof OLDVAL))
      (ASSERT ([y location] flag $)(y location ?)
              ([p locationof] flag $)))
     ((y | (y elementof OLDVAL) ~(y elementof NEWVAL))
      (ASSERT (y location p) ~([y location] flag $))
      (((  ∀z ) (p locationof z) ⇒(z holding NIL))
       (ASSERT ~([p locationof] flag $)]                    (2.72)

ATR[PLACE locationof] =
[((T ?)(((∀y)(y elementof NEWVAL) ⇒ (y holding NIL))
       (ASSERT ~([@ locationof] flag $))
       (((s elementof NEWVAL) ~(s holding NIL))
        (ASSERT ([@ locationof] flag $))))
(NIL ((x | (s holding x)(s elementof NEWVAL)
           (x type FIXED))
     (ASSERT ~(s holding x))                                (2.73)
```

Also, let us associate with [Y location] for Y = HUMAN, VEHICLE, VEGETABLE, ANIMAL, the following CC:

```
CC[HUMAN location] = CC[VEGETABLE location] =
  CC[ANIMAL location] = CC[VEHICLE location] =
  [(PLACE p) | (Q location p)(Q heldby NIL) V
               (Q heldby:location p)]                       (2.74)
```

The ATR in (2.72) does the following:

For all y such that @ has just gotten hold of the y -- i.e., (y elementof NEWVAL) ~(y elementof OLDVAL) -- the dummy flag, $, is asserted for the invocation anchor [y location]. Also (y location) = ? is set, and for the PLACE, p, such that (@ location p), ([p locationof] flag $) is asserted. As discussed in section (2.3.2), item (iii), the $ flag has the following interpretation: Every time (y location) is called for, its value will be computed using the CC and/or ATR associated with [y location]. In our case, this would cause the CC in (2.74) to be evaluated, and (@ heldby:location) to be returned. In the case of [p locationof], the value of (p locationof) may be, (x_1 x_2 ... ?), where the ? indicates the presence of possibly more elements in the collection. The CC associated with [p locationof] will be evaluated to find these additional elements, namely the elements that are being held by x_1, x_2 ... .

The second part of the ?-T-actions in (2.72) takes care of the case when an object is just let go off by @ --- i.e. ~(y elementof NEWVAL)(y elementof OLDVAL). For all these y, (y location p) is asserted. That is, y is put at the place at which @ is located. The $ flag on [y location] is removed, and if none of the objects at p is holding anything then the $ flag at [p locationof] is also removed. This sets the representation back to old form.

The ATR[PLACE locationof] keeps track of moving the flag as an object at a PLACE, that is holding something, moves

to another place. If x is at p, and (x holding m) is true, then by (2.72), ([p locationof] flag $) will be true. Now, if x moves to g, then ([g locationof] flag $) is asserted, and if p does not have any more objects that hold something then ~([p locationof] flag $) is asserted.

The selective association of these $ flags now completely eliminates the need for secondary updating in the model space, as locations change. Thus, a VEHICLE may be holding a hundred passengers. But as it moves, the only secondary change in the model space will be the movement of the $ flag associated with [p locationof]. The locations of the passengers themselves need not be changed.

In the example we discussed, fortunately, there was no propagation of secondary changes in the updating process. The propagation characteristics at an [X r] are governed by the, so called CLOSURE([X r]),

$$\begin{aligned} &\text{CLOSURE}([X\ r]) = \\ &\qquad (\text{DONLIST}[x\ r] \cup \text{DETLIST}[X\ r] \cup \\ &\qquad (\text{CLOSURE}(\text{DONLIST}[X\ r] \cup \text{DETLIST}[X\ r])) \end{aligned} \tag{2.75}$$

where the closure of an union is the union of the closures of its elements. In the case of [PLACE locationof] , we had

$$\text{CLOSURE}([\text{PLACE locationof}]) = \text{DONLIST}[\text{PLACE locationof}]. \tag{2.76}$$

This was the reason, why we had no propagation of secondary changes,

One may define the depth of an [X r] to be the number of applications of the CLOSURE operator in equation (2.75). For [PLACE locationcf], its depth is 1. In general, the definitional filters and the focus lists may be ordered at an [X r] in increasing order of the depth of [Y t], for [Y t] ϵ DETLIST [X r], and [Y t] ϵ DONLIST[X r]. In an updating process, one may choose the secondary changes, in increasing order of their depths, preferring those with smaller depths over those with larger depths.

CLOSURE[X r] may be computed from domain definitions. One may also, of course, compute a CLOSURE[$\mathcal{C}_X$ r] for an instance [$\mathcal{C}_X$ r] of [X r], at "run time" by using the definitional filters.

This completes the discussion of the basic modelling concepts in MDS. We shall conclude this section with a review of what we have done so far.

2.5: Object Based Representations: Bundles.

We have proposed a way of defining relational systems, R, and using them for two purposes: One is to define the syntax and semantics of elementary description languages, L, that are based on R. The other is to define the structures and processes of model spaces, M.

The bundle structures are obtained by viewing the relational schemas and their consistency from the point of view of classes, X, that naturally occur in the domain: what relations may occur with X, and on what classes, Y, do the relations of X depend on. The bundle of an entity, x, is a representation in the model space, of the description of x in the language L. It is a model of x not only in the sense that it is a representation of x, but also in the sense that it satisfies within the 3-valued logical system all the properties of logical consistency, that instances of the class X should themselves satisfy, in the relational system R.

Slots in the bundle not only have values, but also the reasons and hypotheses associated with the values. Where a value is unknown, it may be characterized by conditions, expressed in L. Most importantly, all the potential interactions of a bundle with other bundles in the model space may be derived economically using the DF and FF filter schemes. The depth of such interactions may be controlled at problem solving time by the use of frame filters, and the filters switch.

The schematic definition of the properties of a class X in R, is also the definition of the bundle scheme, used to represent members of class X. What makes these schematic definitions useful and interesting is that they are instantiable. Thus, the bundle paradigm, which is essentially a description paradigm may also be used as a programming paradigm: What is described as a bundle schema may also be instantiated in the model space. Of course, it is assumed that the schema definitions themselves are consistent.

To instantiate, X, one may first create a new instance of the data-type that is associated with X, if the CC[X instance] and ATR[X instance] associated with X, admit of such an instance. Initially, all the slots of $@_X$ will contain, ?, indicating that their values are all unknown. The slots may be filled by issuing (INSTANTIATE ($@_X$ r)) commands for each relation r, that is associated with $@_X$. If CC[X r] is imperative, then this may succeed in finding the (y | ($@_X$ r y)), or characterizing this value by a conditinn. Otherwise, it may find the candidates for ($@_X$ r). The choices of y for ($@_X$ r y) would then have to be made by the DESIGNER, TP or a user, depending on the context of creation of $@_X$.

Modification to the model space are done by the INSTANTIATE, FORCE and DELETE commands. The problem solving systems or a user that issue these commands need not, however, be too domain specific. The ASSIMILATOR can use the domain knowledge to understand

the consequences of a given assertion, and where there is an inconsistency, provide the reasons for it. This frees the programmer or the problem solver from a whole class of details of the domain characteristics, namely all those that pertain to the consistency of the model space. Thus programs can be vague or may even contain errors. The system can respond intelligently to unexpected situations. We shall refer to this kind of programming as Knowledge Based Programming Srinivasan [1973b, 1977c], a programming methodology in which a user or a system need not be aware of all the relevant domain laws. The paradigm for knowledge based programming is illustrated in figure 3. The program control uses the reasons and hypotheses supplied by the ASSIMILATOR to decide on the next step in program execution. Also, these reasons and hypotheses are used to update the program execution state. It should be noted that in this paradigm the

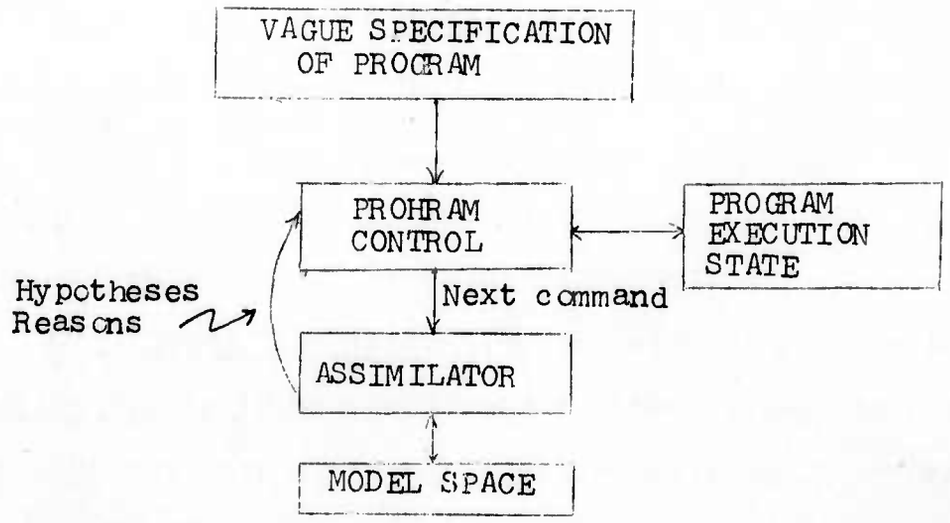

Fig. 3: Paradigm for Knowledge Based Programming

"next command" generated by the program control, may not be present in the vague specification of the program. It may be one that is generated by the control.

The organization of the program control and the program execution state is a characteristic of the kind of problem solving that is done by the program, to decide on the next command. In MDS we distinguish between three general schemas for doing this: The goal directed schema of DESIGNER, the model construction schema of THEOREM PROVER, and the understanding schema of LINGUIST. Thus, in the MDS paradigm, an intelligent machine will have four execution controls, with associated data organizations: Those of ASSIMILATOR, DESIGNER, TP and the LINGUIST.

This kind of use of model space is made possible largely because of the choice of object-based representations, as the basis for defining relational systems, and the use of anchored CC's for constraint specification. In this schema the CC's are used both as functions and as predicates. The advantages of object-based representations, as contrasted with operator-based representations, are summarized below:

(a) Local Isolation: The effects of an inconsistency at a definition anchor is localized to itself, and other relations that interact with it. The interactions themselves are predictable and smoothly controllable, using filters and focus lists. Errors can be characterized in terms of the static properties of

of the model space. In operator-based representations, error propogation characteristics cannot often be easily stated in terms of the static properties of a model space. They may be characterized only in terms of operator sequences, or a grammar. These are not convenient entities with which one may reason.

(b) Immediacy and Focus: The items that participate in a predicate, function or a problem solving process can provide immediate access to all properties, constraints, transformatins, and combinations of reasons, that might be relevant to the context of their use. In operator-based representations, the interactions among items will depend on the operator sequence involved, and combinations of properties relevant to a task may only be indirectly obtained via the effects produced by the operator sequences.

(c) Flexibility: This is partly a consequence of (a) above. Changes and extensions to definitions of classes in a domain may be introduced more gracefully in object-based schemas, than in operator-based ones. Changes and extensions in an operator-based scheme may call for a change of the entire system.

(d) Incompleteness: In object-based schemes, incomplete data result in partitioning of properties into known and unknown categories, where the 3-values logical system may be uniformly used to characterize a model space. In operator-based schemes, unless the operators are defined a priori to account for every

possible combination of unknowns, it is not possible to characterize conveniently the state of a model space. Unknowns in operator-based systems will result in "non-determinism", which may contribute to combinatorial explosions.

In the next chapter we shall present the basis for commonsense reasoning and briefly discuss algorithms for CC-evaluation, and residue extraction.

III Residues, Commonsense Reasoning and CC-evaluations

In this chapter we shall define the syntax of CC's, define the ξ-calculus that is used to compute residues and prove the properties of residues that make them useful for commonsense reasoning. The CC-evaluation process and associated representations are discussed in section 3.4. Some of the representation schemes in the MDS model space, that facilitate CC-evaluation are discussed in Appendix II.

The methods discussed in this section are not unique in any sense. They are included here only to make the definitions precise and to point out the feasibility of using the commonsense reasoning paradigm. In a subsequent paper we shall discuss the complexity of CC evaluation processes, in the context of the MDS model space.

3.1: Syntax of Consistency Conditions

```
<CC>       →  <set-exp>
<set-exp>  →  (<set-var> "|"<P>)
<set-var>  →  <var> | <tuple> |
              (<scope>␣<var>)
              (<scope>␣<tuple>)
<scope>    →  <class> | <class>/<scope>
              <class>\<scope> | λ
```

<class> is used to denote the name of a class in a domain. It is also the name of template that defines the class. The symbol "␣" is used for blank. λ is used for null string. It is also

later used for denoting null sets

<var> → x | y | $x_{<i>}$ | $y_{<i>}$ |

<i> → 0 | 1 | 2 | ...

<v> → <var> | <tuple> | <fn-call>

<tuple> → []

<p-arg> → <var> | (<v> ␣ <r>) | (<r> ␣ <v>)

 → <p-arg> | ␣ <p-arg>

<r> → <relname> | <relname>:<r>

<relname> is used to denote a relation name in the domain.

<fn-call> → (<fn-name> ␣ <args>)

<fn-name> denotes the name of a function defined by using the function template schema. Functions appearing in a CC are required not to change the model space during CC evaluation.

<args> → | <set-exp> |
<args> ␣ <args>

<quantifier> → (<scope> ␣ <b-vars>) |
(<q-type> ␣ <scope> ␣ <b-vars>) |
(SOME ␣ <scope> ␣ <var> ␣ <i> ␣ <i>)

<q-type> → ALL | SOME | THE

In our discussions in this chapter we will use "(∀x)" and "(∃x)" as the symbols for quantification. This is done only for convenience. All quantifiers in CC's are required to be as specified by <quantifier>. In the existential quantification, "(SOME ... <i> <i>)" the integers are used to indicate lower and upper bounds on the number of solutions.

<b-vars> → <var> | <var> ␣ <b-vars>
<p> → (<p-arg> ␣ <r> ␣ <p-arg>) |
(<tuple-name> ␣) |
~ <p>

<p> is the elementary predicate. We shall use symbols p, q, pi, qi, etc. to denote these. Notice that <p> can appear with or without negation. The <tuple-name> denotes the name of a tuple template. Tuple templates are used in MDS to define n-ary relations for $n \geq 1$.

<&-seg> → <P> | <P> ∧ <&-seg>
<&-exp> → (<&-seg>) | <P> |
<P> <&-exp>
<V-seg> → <P> | <P> V <V-seg>
<V-exp> → (<V-seg>)
<⇒-exp> → (<P> ⇒ <P>)
<⇐⇒-exp> → (<P> ⇐⇒ <P>)
<~exp> → ~<P> | (~<P>)
<q-seg> → <quantifier> <P> |
<quantifier> <q-seg>

We shall use the symbol <q> to denote a quantifier.

<L-exp> → <p> | <&-exp> | <V-exp> |
< ⇒-exp> | <⇐⇒-exp) | <~exp>
<P> → <L-exp> | (<q-seg>).

The following special symbols are used, whenever convenient

$$
\begin{array}{rcl}
* & \rightarrow & V \mid \wedge \\
*\lambda & \rightarrow & \lambda \\
\langle P \rangle\ \lambda & \rightarrow & \langle P \rangle \\
\lambda\ \langle P \rangle & \rightarrow & \langle P \rangle \\
\xi & \rightarrow & T \mid U \mid F
\end{array}
$$

Thus ξR will denote TR, UR and FR, φ is used throughout as the evaluation function, and α is used to denote a substitution list of the form

$$\alpha = [(x\ a)(y\ b)\ \ \ldots]$$

indicating substitution of a for x, b for y, etc. $[\alpha]P$ is the predicate expression P with substitution α. $\varphi[\alpha]P$ is the truth value of $[\alpha]P$. By convention $\varphi_{\lambda} = ?$, $\varphi\xi$ will denote T, ? or NIL.

Consistency conditions are represented in the model space in their mini-scope forms. This form is defined in the next section, and rules for converting an arbitrary P to its mini-scope form are presented.

3.2 The Mini-Scope Form

Let $\langle\forall\rangle$ denote a string of universal <quantifiers>, and $\langle\exists\rangle$, a string of existential <quantifiers>. Let Px be a predicate expression in which x occurs free, and $P\bar{x}$ be one in which x does not occur free. Then, the definition of the Mini-Scope Form (MSF) may be stated as follows:

(F1) p is in MSF

(F2) If P and Q are in MSF then so are (P v Q), PQ, (PQ), (P$\wedge$Q) and P$\wedge$Q.

(F3) If P is in MSF then $((\forall x)P)$ is in MSF only if x occurs free in P, and P is not of the form $P_1 P_2$, $(P_1 \wedge P_2)$

(F4) If P is in MSF then $((\exists x)P)$ is in MSF only if x occurs free in P, and P is not of the form $(P_1 \vee P_2)$.

The following rules may be used to convert an arbitrary P to its MSF. The propositional rules given below are to be applied first. The rules are applied until no more rules can be used.

(A) Propositional Rules

i) $\sim \sim P \rightarrow P$

ii) $(P \Rightarrow Q) \rightarrow (\sim P \vee Q)$

iii) $(P \Leftrightarrow Q) \rightarrow (PQ \vee \sim P \sim Q)$

iv) $\sim(PQ) \rightarrow (\sim P \vee \sim Q)$

v) $\sim(P \vee Q) \rightarrow \sim P \sim Q$

(B) Quantificational Rules

i) $\sim((\forall x)P) \rightarrow ((\exists x)\sim P)$

ii) $\sim((\exists x)P) \rightarrow ((\forall x)\sim P)$

iii) $((\forall x)\ P\bar{x} \rightarrow P$

iv) $((\exists x)\ P\bar{x}) \rightarrow P$

Here $P\bar{x}$ indicates that x does not occur free in P.

v) $(\langle\forall\rangle PQ \rightarrow (\langle\forall\rangle P)(\langle\forall\rangle Q)$

vi) $(\langle\exists\rangle(P \vee Q)) \rightarrow ((\langle\exists\rangle P) \vee (\langle\exists\rangle Q))$

viii) $((\forall x)\ \langle\forall\rangle\ (Px \vee Q\bar{x})) \rightarrow$
$(\langle\forall\rangle\ (((\forall x)P) \vee Q)$

viii) $((\forall x)\langle\forall\rangle\ (P\bar{x}\ V\ Qx) \rightarrow (\langle\forall\rangle\ (P\ V\ ((\forall x)Q)))$

ix) $((\exists x)\langle\exists\rangle\ Px\ Q\bar{x} \rightarrow (\langle\exists\rangle\ ((\exists x)P)Q)$

x) $((\exists x)\langle\exists\rangle\ P\bar{x}\ Qx \rightarrow ((\langle\exists\rangle\ P\ ((\exists x)Q)).$

We shall assume that the variables in a predicate expression are all distinct. In the current implementation of MDS, the program for transforming CC's to their MSF was written by Tau Hsu.

In the next section the residues are defined and the ξ-calculus that is used for residue extraction is introduced.

3.3 Residues and partitions.

We shall first define residues for propositional expressions. As mentioned before, let α be a substitution list of the form:

$$\alpha = [\,(x\ a)(y\ b)\ \ldots\ ] \tag{3.1}$$

If x is in α, then we shall let

$$[\alpha]((\forall x)P) = [\alpha]((\exists x)P) = [\alpha]P. \tag{3.2}$$

If x is free in P and x does not occur in α then we shall assign x = ? in P. Also we have,

(a) $\varphi(x\ r\ ?) = ?$ for all x for which (x r) is dimensionally Consistent.

(b) For a tuple name R,

$$\varphi(R\ a_1\ a_2\ \ldots\ ?\ \ldots\ a_n) = T \text{ if}$$
$$\varphi((\forall x)(R\ a_1\ a_2\ \ldots x \ldots a_n)) = T$$
$$= NIL \text{ if}$$
$$\varphi((\forall x)(R\ a_1\ a_2 \ldots x \ldots a_n)) = NIL$$
$$\text{else} \quad = ?$$

(c) $\varphi(?\ r) = \varphi(r\ ?) = ?$

where r is a relation path

For a function, f, if one or more of its arguments is unknown then its value is ?, unless the function itself returns another value. Thus, in the 3-valued logical system $\varphi[\alpha]P$ is always well defined.

We shall require that expressions of the form, $(\varphi P = ?)$, do not occur in CC's. This is consistent with our view that in a domain itself, there is no concept of a relation having value, ?: Every relation is NIL or T in some model space. However, there may exist P, for which it may be impossible to construct the model space in which P = T or NIL. In MDS, such predicates may have value ?, and computations attempting to assign a T or NIL value to them, may never terminate. This view is consistent with semi-decidability of sentences in first order logic.

3.3.1 Residues and Partitions of Propositions

We shall assume that all propositions are in mini-scope form: i.e. negations appear only in the elementary forms p, and $\wedge$ and V are the only connectives. We shall consider below, only grounding substitutions, i.e., P has no variables in it.

Definition 3.1: Residues of Elementary Forms

$$(\xi R[\alpha]p = p) \quad \text{if } \varphi[\alpha]p = \varphi\xi, \quad \text{else } \lambda \quad \ldots \tag{3.4}$$

$$\begin{aligned} TR[\alpha]{\sim}p &= {\sim}FR[\alpha]p \\ FR[\alpha]{\sim}p &= {\sim}TR[\alpha]p \\ UR[\alpha]{\sim}p &= {\sim}UR[\alpha]p \end{aligned} \tag{3.5}$$

Definition 3.2: Residues of Conjunctions and Disjunctions

Let * be $\wedge$ or V. Then

$$\xi R[\alpha](P_1 * P_2 * \ldots * P_k) = \overset{k}{\underset{i=1}{*}}(\xi R[\alpha]P_i) \tag{3.6}$$

Lemma 3.1

$$(\varphi[\alpha]P \neq \varphi\xi) \Rightarrow (\xi R[\alpha]P = \lambda) \tag{3.7}$$

Lemma 3.2

$$(\xi R[\alpha]P \neq \lambda) \Rightarrow (\varphi[\alpha](\xi R[\alpha]P) = \varphi\xi) \tag{3.8}$$

Proofs of these lemmas are by induction on the structure of P: P is a <p> or P is $(P_1 \vee P_2 \vee \ldots \vee P_k)$ or $P_1\ P_2 \ldots P_k$.

We shall hereafter indicate explicitly the bindings associated with a residue, by writing the residue in the form

$$[\alpha](\xi R[\alpha]P).$$

3.3.2 Residues and Partitions of Predicate Expressions

3.3.2.1 Substitution Ranges, Their Partitions and Solutions of P

A substitution range is a list of the form

$$\underline{\alpha} = (\underline{\alpha}^1\ \underline{\alpha}^2\ \ldots\ \underline{\alpha}^n) \tag{3.9}$$

where each

$$\underline{\alpha}^i = [(x_1\ (a^i_{11}\ a^i_{12}\ \ldots))(x_2(a^i_{21}\ a^i_{22}\ldots))\ \ldots\ (x_m\ (a^i_{m1}\ a^i_{m2}\ \ldots\ ))],\ldots \tag{3.10}$$

in which a^i_{jk} are constants. For $i = 1,2,\ldots,n$, we shall say that

$$(\text{Range}\ \underline{\alpha}_i\ x_j) = (a^i_{j1}\ a^i_{j2}\ \ldots) \tag{3.11}$$

Definition 3.3: $(\underline{\alpha} \supseteq \underline{\beta})$ and $(\alpha \in \underline{\alpha})$.

(i) $(\alpha \in \underline{\alpha}^i)$ if α and $\underline{\alpha}^i$ have the same variables, and for each x in α,

$(\text{Range}\ \alpha\ x) \subseteq (\text{Range}\ \underline{\alpha}^i\ x)$.

(ii) $(\alpha \in \underline{\alpha})$ if there exists an i such that $(\underline{\alpha}^i$ memberof $\underline{\alpha})$ and $\alpha \in \underline{\alpha}^i$.

(iii) $(\underline{\alpha} \supseteq \underline{\beta})$ or $(\underline{\beta} \subseteq \underline{\alpha})$ if

$$((\forall\delta)(\delta \in \underline{\beta}) \Rightarrow (\delta \in \underline{\alpha})) \qquad (3.12)$$

In the context of predicate expressions, we shall at times distinguish between three kinds of substitution ranges.

1) Range of Universals: $\underline{\alpha}_u(P)$

$\underline{\alpha}_u(P)$ is of the form

$$\underline{\alpha}_u(P) = ([(x(a_1 a_2 \ldots))(y(b_1 b_2 \ldots))\ldots \ldots]) \qquad (3.13)$$

and it specifies the range of values for the universally quantified variables of P.

(ii) Free Range: $\underline{\alpha}_f(P)$

$\underline{\alpha}_f$ is of the form (3.9), and it specifies the range of values for the free variables of P. If P has no free variables, then $\underline{\alpha}_f(P) = \lambda$.

(iii) Solution Range: $\underline{\alpha}_s(P\ \underline{\alpha}_f)$

$\underline{\alpha}_s$ is of the form (3.13) and it specifies the solution for the existentially quantified variables of P, for the given free range $\underline{\alpha}_f$.

Since we have assumed that all variables in P are distinct clearly, $\underline{\alpha}_u$, $\underline{\alpha}_f$ and $\underline{\alpha}_s$ will not have any common variables. By definition, let us set

$$[\lambda]P = P \quad \text{and} \quad (\lambda \in \underline{\lambda})$$

We shall often use $\underline{\alpha}^x$ or α^x to denote a substitution range in which x is the only variable. We shall use $\xi\underline{\alpha}_s(P\ \underline{\alpha}_f)$ to denote the ξ-solution of P for the free range $\underline{\alpha}_f$. Also, we shall generally use expression of the forms:

$$([\xi\underline{\alpha}_f][\xi\underline{\alpha}_u][\xi\alpha_s]P)$$

with the interpretation

$$(\forall\alpha)(\alpha \in \xi\underline{\alpha}_f)(\forall\beta)(\beta \in \xi\underline{\alpha}_u) \Rightarrow$$
$$((\exists\delta)(\delta \in \xi\underline{\alpha}_s)(\varphi[\alpha][\beta][\delta]P = \varphi\xi) \qquad (3.14)$$

irrespective of, whether the variables in $\underline{\alpha}_u$ are universally quantified, and those in $\underline{\alpha}_s$ are existentially quantified. But, always $\underline{\alpha}_u$ and $\underline{\alpha}_s$ are used to specify ranges for the bound variables of P.

We will consider residues of predicates, P, only in the context of a given $\underline{\alpha}_f$. We shall make use of the fact that P is in mini-scope form, and define the residues only for expressions of the form $((\forall x)(P_1 \vee P_2 \vee \ldots \vee P_k))$ and $((\exists x)(P_1 P_2 \ldots P_k))$, among quantified expressions.

Every predicate expression P, may be reduced to its equivalent propositional form in the model space, since P is evaluated only over the models that exist in the model space. Thus, residues of P are equivalent to residues of their associated propositional forms. But, propositional residues of this kind are likely to be very large expressions. They are hard to generalize. They cannot easily be used in new situations, when the bindings of the variables change. The expressive power of the sentential forms of P is completely lost in the propositional residues. We shall discuss below a way of extracting the residues in which the sentential forms of P are not lost. We shall express the residues in terms of expressions of the forms:

$$\xi R(P) = [\langle \text{scope of bindings for variables in } Q \rangle]Q.$$

where Q would be a subexpression of the parent expression P, and the bindings specify the context for Q. If the model space changes and the bindings change, then Q may be easily re-evaluated in the context of the new bindings. Since Q is, in general, a sub-expression of P, the re-evaluation of Q will always be a simpler task, than the re-evaluation of P.

3.3.2.2. ξ-Solutions of P

Definition 3.4: Partitions Induced by P

Every predicate expression, P, induces a partition of the

range of its free variables, $\underline{\alpha}_f(P)$ into three parts $\xi\underline{\alpha}_f(P)$ for $\xi = T, U$ and F:

$$\xi\underline{\alpha}(P) = (\alpha \mid (\alpha \in \underline{\alpha}_f(P))(\varphi[\alpha]P = \varphi\xi) \tag{3.15}$$

Also, if z is the only bound variable of P, then we shall say that,

$$\xi z(P\ \alpha) = (u \mid (\varphi[\alpha][(z\ u)]P = \varphi\xi)) \tag{3.16}$$

and

$$z(P\ \alpha) = [Tz(P\ \alpha)\ Uz(P\ \alpha)\ Fz(P\ \alpha)] \tag{3.17}$$

If P is an elementary predicate, p, then $z(p\ \alpha)$ may be computed using the model space. For binary relations, the storage of relation values in terms of collections (See Appendix II), facilitates the direct retrieval of $z(p\ \alpha)$ from the model space. For n-ary relations, $n > 2$, the partition $z(p\ \alpha)$ will be computed when called for.

We shall present the residue definitions in such a manner, that it makes apparent its computation. Besides computing residues, we shall also compute the, so called, partitions of P for a given $\underline{\alpha}_f$. An example of a partition of P is presented below:

Suppose $\varphi[\underline{\alpha}_f]P = ?$ and $P = ((\forall z)P_1)$. Then by definition

$$((\exists\alpha)(\alpha \in \underline{\alpha}_f)(\varphi[\alpha]P = ?))$$
$$\sim(\exists\beta)(\beta \in \underline{\alpha}_f)(\varphi[\beta]P = \text{NIL}) \quad \ldots \quad (3.18)$$

Thus, in this case $F\underline{\alpha}_f(P) = \lambda$. However, $T\underline{\alpha}_f(P)$ may exist. $U\underline{\alpha}_f(P)$ would, of course, exist. The true residue will have the form

$$TR[T\underline{\alpha}_f]P = [T\underline{\alpha}_f(P)][T\underline{\alpha}_u][T\underline{\alpha}_s]Q \quad \ldots \quad (3.19)$$

for some sub-expression Q of P. For each α in $U\underline{\alpha}_f(P)$ the universally quantified variable, z, may have its range partitioned into two parts:

$$Uz(P\ \alpha) = (b_1\ b_2\ \ldots\ b_m\ ?) \quad \text{and}$$
$$Tz(P\ \alpha) = (a_1\ a_2\ \ldots\ a_n\ ),$$

such that for $\xi = U$ and T

$$(u \in \xi z(P\ \alpha)) \Rightarrow (\varphi[\alpha][(z\ u)] = \varphi\xi) \quad (3.20)$$

In this case, we shall say that

$$U\underline{\alpha}_f(P) = ([\alpha\ (z\ Uz(P\ \alpha))]\ |\ (\alpha \in U\underline{\alpha}_f(P))(Uz(P\ \alpha) \neq \lambda)] \quad (3.21)$$

and $$UR[\underline{\alpha}_f]P = UR[U\underline{\beta}_f(P)]P_1 \quad (3.22)$$

Also, $$U_{T\underline{\beta}_f}(P) = ([\alpha\ (z\ Tz(P\ \alpha))\ |\ (\alpha \in U\underline{\alpha}_f(P))(Tz(P\ \alpha) \neq \lambda)] \quad (3.23)$$

and the true partition of P,

$$\Pi T[\underline{\alpha}_f]P = (TR[T_{\underline{\alpha}_f}(P)]P \wedge TR[U_{T^{\underline{\beta}_f}}(P)]P_1) \qquad (3.24)$$

This is the part of P with its associated bindings, that evaluated to true, when P itself evaluated to ? for $\underline{\alpha}_f$.

True part of a P may also exist where $\varphi[\underline{\alpha}_f]P = NIL$. In this case one may also have the unknown part of P. Even when P is a proposition, one may have a true part (unknown part) of P, when P evaluates to ? (NIL). Thus, $P = P_1P_2 \ldots P_k$ may evaluate to ? for an α, but some P_i, for $1 \leqslant i \leqslant k$ may be such that $\varphi[\alpha]P_i = T$.

We shall throughout use the convention that

$$\xi R[\lambda]P = \Pi\xi[\lambda]P = \lambda \qquad (3.25)$$

Also, in binding specifications we shall omit the value ranges for universally quantified variables, if the range is equal to all the instances in the scope of the variable. In cases like this we shall usually say, "the range of x is equal to the scope of x".

In the subsection below we define the, so called, ξ-calculus that is used to compute ξR-residues and $\Pi\xi$-partitions of P for given ranges of $\underline{\alpha}_f$. The calculus is defined inductively on the structure of P:

(i) P is a p or a $\sim$p.

(ii) $P = (P_1 * P_2 * \ldots * P_k)$, $k \geq 1$, $* = \wedge$ or $\vee$.

or (iii) $P = ((\forall y)(P_1 \vee P_2 \vee \ldots \vee P_k))$ or
$((\exists y)\, P_1 P_2 \ldots P_k)$, $k \geq 1$,

where P_i, $1 \leq i \leq k$ are general predicate expressions in mini-scope form.

Definition 3.5: Truth Value of $[\underline{\alpha}_f]P$.

$$((\varphi[\underline{\alpha}_f]P = T) \Leftrightarrow ((\forall\alpha)(\alpha \in \underline{\alpha}_f) \Rightarrow (\varphi[\alpha]P = T)) \quad (3.26)$$

$$((\varphi[\underline{\alpha}_f]P = ?) \Leftrightarrow ((\exists\alpha)(\alpha \in \underline{\alpha}_f)(\varphi[\alpha]P = ?))$$
$$\sim((\exists\beta)(\beta \in \underline{\alpha}_f)(\varphi[\beta]P = NIL)) \quad (3.27)$$

$$(\varphi[\underline{\alpha}_f]P = NIL) \Leftrightarrow ((\exists\alpha)(\alpha \in \underline{\alpha}_f)(\varphi[\alpha]P = NIL)) \quad (3.28)$$

In what follows we shall consider all propositions and predicate expressions to be (n+2)-ary predicates for $n \geq 0$: Elementary forms $(P\ x_1 \ldots x_n\ y\ z)$, propositions $(M\ x_1 \ldots x_n\ y\ z)$ $(M_1 M_2 \ldots M_k)$, $(N\ x_1 \ldots x_n\ y\ z)$, $(N_1 \vee N_2 \vee \ldots \vee N_k)$, etc. We shall throughout assume that the solutions

$$\xi_{\underline{\beta}_f}(Z) = ([x_1 \ldots x_n\ y\ z] \mid (\varphi(Z\ x_1\ x_2 \ldots x_n\ y\ z) = \varphi\xi) \quad (3.29)$$

is available to us for a general expression Z. In section 3.5 we shall briefly outline a procedure for obtaining $\xi_{\underline{\beta}_f}(Z)$ for a proposition Z, using the relation values stored in the model space.

3.3.3: Elementary Form

$$P = (p\ x_1 \ldots x_n\ y\ z) \quad \text{or}$$
$$\sim(p\ x_1 \ldots x_n\ y\ z) \tag{3.30}$$

$\xi\underline{\beta}_f(P)$ may be obtained from the model space. For a given α, $\xi z(P\ \alpha)$ may be directly retrieved from the model space, as also the partition $z(P\ \alpha)$ (see equations (3.16) and (3.17)): The partitions of P,

$$\Pi\xi[\underline{\alpha}_f]P = [\xi\underline{\beta}_f(P)]P \tag{3.31}$$

where

$$\xi\underline{\beta}_f(P) = ([\alpha\ (z\ \xi z(P\ \alpha))]\ |\ (\alpha \in \underline{\alpha}_f)\ (\xi z(P\ \alpha) \neq \lambda)) \tag{3.31a}$$

$$\xi R[\xi'\underline{\beta}_f(P)]P = [\xi\underline{\beta}_f(P)]P \text{ if } \xi' = \xi \text{ else } \lambda. \tag{3.32}$$

We shall hereafter uniformly use the symbol β to denote a binding for $[x_1 \ldots x_n\ y\ z]$ and the symbol, α, for a binding of the prefix $[x_1 \ldots x_n\ y]$. For any proposition, Z, we shall hereafter use the solutions defined below:

(A) $$\xi\underline{\alpha}_f(Z) = (\alpha\ |\ (\xi z(Z\ \alpha) \neq \lambda)) \tag{3.33}$$

(B) $$UF_T\underline{\alpha}_f(Z) = (\alpha\ |\ (\alpha \in UF\underline{\alpha}_f(Z))\ (Tz(Z\ \alpha) \neq \lambda)) \tag{3.34}$$

(C) $$UF\underline{\alpha}_f(Z) = (U\underline{\alpha}_f(Z) \cup F\underline{\alpha}_f(Z)) \tag{3.35}$$

(D) $$F_U\underline{\alpha}_f(Z) = (\alpha\ |\ (\alpha \in F\underline{\alpha}_f(Z))\ (Uz(Z\ \alpha) \neq \lambda)) \tag{3.36}$$

Note that an α can be in all of the solutions $\xi_{\underline{\alpha}_f}(Z)$.

(E) $$\xi_{\underline{\beta}_f}(Z) = ([\alpha\ (z\ \xi z(Z\ \alpha))]\,|\,(\alpha \in \xi_{\underline{\alpha}_f}(Z))] \tag{3.37}$$

(F) $$UF_{T\underline{\beta}_f}(Z) = ([\alpha\ (z\ Tz(Z\ \alpha))\,|\,(\alpha \in UF_{T\underline{\alpha}_f}(Z))] \tag{3.38}$$

(G) $$F_{U\underline{\beta}_f}(Z) = ([\alpha\ (z\ Uz(Z\ \alpha))\,|\,(\alpha \in F_{U\underline{\alpha}_f}(Z))] \tag{3.39}$$

All the above solutions may be computed if $\xi_{\underline{\beta}_f}(Z)$ is known.

3.3.4: Propositional Forms

$N = (N_1 \vee N_2 \vee \ldots \vee N_k)$ and

$M = M_1 M_2 \ldots M_k$, with arguments

$[x_1\ x_2 \ldots x_n\ y\ z]$.

We have the following definitions:

(a) $$Tz(N\ \alpha) = \bigcup_{i=1}^{k} Tz(N_i\ \alpha)$$

(b) $$Uz(N\ \alpha) = \bigcup_{i=1}^{k} Uz(N_i\ \alpha) - Tz(N\ \alpha)$$

(c) $$Fz(N\ \alpha) = \bigcap_{i=1}^{k} Fz(N_i\ \alpha)$$

(d) $$Tz(M\ \alpha) = \bigcap_{i=1}^{k} Tz(M_i\ \alpha)$$

(e) $$Uz(M\ \alpha) = \bigcup_{i=1}^{k} Uz(M_i\ \alpha) - Fz(M\ \alpha)$$

(f) $$Fz(M\ \alpha) = \bigcup_{i=1}^{k} Fz(M_i\ \alpha) \tag{3.40}$$

Knowing the solutions in (3.40) for each $\alpha \in \underline{\alpha}_f$ one may then compute all the solutions (A) through (G) given in the previous section

Definition 3.6: Residues of Propositions with mini-scope expressions.

If $P = (P_1 * P_2 * \ldots * P_k)$, where $* = \wedge$ or V, and each P_i is a mini-scope expression, then let

$$\xi\underline{\alpha}_f^i(P) = (\xi\underline{\alpha}_f(P) \cap \xi\underline{\alpha}_f(P_i)) \tag{3.41}$$

$$\text{Then } \xi R[\underline{\alpha}_f]P = (\overset{k}{\underset{i=1}{*}} (\xi R[\xi\underline{\alpha}_f^i(P))]P_i)) \tag{3.42}$$

If $\xi\underline{\alpha}_f^i(P) = \lambda$ then its asscciated P_i will get dropped off the residue expression.

Definition 3.7: Partitions of Propositions

Again $P = (P_1 * P_2 * \ldots * P_k)$

$$UF_T\underline{\alpha}_f^i(P) = (T\underline{\alpha}_f(P_i) \cap UF\underline{\alpha}_f(P)) \tag{3.43}$$

$$F_U\underline{\alpha}_f^i(P) = (U\underline{\alpha}_f(P_i) \cap F\underline{\alpha}_f(P)) \tag{3.44}$$

$$\Pi T[\underline{\alpha}_f]P = (\overset{k}{\underset{i=1}{*}} (TR[T\underline{\alpha}_f^i(P)]P_i \wedge \Pi T[UF_T\underline{\alpha}_f^i(P)]P_i)) \tag{3.45}$$

$$\Pi U[\underline{\alpha}_f]P = (\overset{k}{\underset{i=1}{*}} (UR[U\underline{\alpha}_f^i(P)]P_i \wedge \Pi U[F_U\underline{\alpha}_f^i(P)]P_i)) \tag{3.46}$$

$$\Pi\xi[\lambda]\ P = \lambda.$$

Using the solutions in (3.40), and definitions 3.6 and 3.7, the residues and partitions of P may be computed. Even

though the solution in (3.40) are stated only for propositions, they apply equally well for forms like

$$P = (P_1 * P_2 * \ldots * P_k)$$

The only requirement is that $[x_1 \ldots x_n \; y \; z]$ be the free variables of P.

3.3.5: Miniscope expressions

3.3.5.1: Universal Quantifier

$$P = ((\forall z)Q)$$

$$Q = (Q_1 \vee Q_2 \vee \ldots \vee Q_k), \quad k \geq 1,$$

Where P has (n+1) free variables $[x_1 \ldots x_n \; y]$ and each Q_i, (n+2) free variables $[x_1 \ldots x_n \; y \; z]$. We have the following:

$$T_{\underline{\alpha}_f}(P) = (\alpha \mid (Tz(Q\ \alpha) = S_p(z))] \tag{3.48}$$

where $S_p(z)$ is the range of values of z in the scope of z, in P.

$$T_{\underline{\beta}_f}(P) = T_{\underline{\alpha}_f}(P) \tag{3.49}$$

We have here integrated out the variable z.

$$U_{\underline{\alpha}_f}(P) = (\alpha \mid (Uz(Q\ \alpha) \neq \lambda)) \tag{3.50}$$

$$U_{\underline{\beta}_f}(P) = [U_{\underline{\alpha}_f}(P)][U_{\underline{\alpha}}{}^{z}(P)] \tag{3.51}$$

where $U_{\underline{\alpha}}{}^{z}(P) = [(z\ (u \mid ((\exists\ \alpha)(u \in Uz(Q\ \alpha)]$ (3.52)

$$F\underline{\alpha}_f(P) = (\alpha \mid (Fz(Q\ \alpha) \not= \lambda)) \tag{3.53}$$

$$F\underline{\beta}_f(P) = [F\underline{\alpha}_f(P)][F\underline{\alpha}^z(P)\ ] \tag{3.54}$$

$$F\underline{\alpha}^z(P) = [(z\ (u \mid ((\exists\alpha)(u \in Fz(Q\ \alpha)))] \tag{3.55}$$

Definition 3.8: Residues of Quantified Expressions for Universal Quantifiers

In this case, none of $\alpha \in \underline{\alpha}_f$ will contain the variable z. We shall extend each α to include z, as follows:

$$\text{Let } Tz^i(P\ \alpha) = Tz(Q_i\ \alpha) \tag{3.56}$$

and for $\xi = U$ and F

$$\xi z^i(P\ \alpha) = (\xi z(Q\ \alpha) \cap \xi z(Q_i\ \alpha)) \tag{3.57}$$

$$\xi\underline{\beta}_f^i(P) = ([\alpha\ (z\ \xi z^i(P\ \alpha))] \mid (\alpha \in \xi\underline{\alpha}_f(P))(\ z^i(P\ \alpha) \not= \lambda)) \tag{3.58}$$

Then,

$$\xi R[\underline{\alpha}_f]P = ((\forall z) \bigvee_{i=1}^{k} (\xi R[\xi\underline{\beta}_f^i(P)]Q_i)) \tag{3.59}$$

Definition 3.9: Partition of P

$$P = ((\forall z)(Q_1 \vee Q_2 \vee \ldots \vee Q_k))$$

In this case $UF_T\underline{\alpha}_f^i(P)$ and $F_U\underline{\alpha}_f^i(P)$ are again as in (3.43) and (3.44), with the exception that "$\xi\underline{\alpha}_f(P_i)$" for $\xi = T$ and U, respectively, should be replaced by "$\xi\underline{\alpha}_f((\exists z)Q_i)$".

$$UF_T\,\underline{\beta}_f^i(P) = ([\alpha\ (z\ Tz^i(P\ \alpha))]\,|\,(\alpha \in UF_T\,\underline{\alpha}_f^i(P))(Tz^i(P\ \alpha) \neq \lambda) \tag{3.61}$$

$$F_U\,\underline{\beta}_f^i(P) = ([\alpha\ (z\ Uz^i(P\ \alpha))]\,|\,(\alpha \in F_U\,\underline{\alpha}_f^i(P))(Uz^i(P\ \alpha) \neq \lambda) \tag{3.62}$$

$$\Pi T[\underline{\alpha}_f]P = (\overset{k}{\underset{i=1}{V}}\ (\exists z)(TR[T\underline{\beta}_f^i(P)]Q_i \wedge TR[UF_T\underline{\beta}_f^i(P)]Q_i)) \tag{3.63}$$

$$\Pi U[\underline{\alpha}_f]P = (\overset{k}{\underset{i=1}{V}}\ (\exists z)(UR[U\underline{\beta}_f^i(P)]Q_i \wedge UR[F_U\underline{\beta}_f^i(P)]Q_i)) \tag{3.64}$$

In this mode of computing residues and partitions the bindings specifying the ξ-solutions for each subexpression of P is passed on to the subexpression, past the quantifier. Ultimately when the subexpression becomes an expression in elementary form, its residues and partitions will be computed as per equations (3.31) and (3.32). At a higher level, a subexpression, Q_i, may get dropped off a residue, or partition expression, if its associated binding is empty.

3.3.5.2: Existential Quantifier

$$P = ((\exists z)(Q_1Q_2\ldots Q_k)),\ k \geq 1,$$

$$Q = Q_1Q_2\ldots Q_k$$

where P has (n+1) free variables $[x_1 \ldots x_n\ y]$ and each Q_i has (n+2) free variables $[x_1, \ldots x_n\ y\ z]$. We have the following:

$$T\underline{\alpha}_f(P) = (\alpha \mid (Tz(Q\ \alpha) \neq \lambda)) \tag{3.65}$$

$$T\underline{\beta}_f(P) = [T\underline{\alpha}_f(P)][T\underline{\alpha}^z(P)] \tag{3.66}$$

$$T\underline{\alpha}^z(P) = [(z\ (u \mid ((\exists\alpha)(u \in Tz(Q\ \alpha)] \tag{3.67}$$

$$U\underline{\alpha}_f(P) = (\alpha \mid (Tz(Q\ \alpha) = \lambda)(Uz(Q\ \alpha) \neq \lambda) \tag{3.68}$$

$$U\underline{\beta}_f(P) = [U\underline{\alpha}_f(P)][U\underline{\alpha}^z(P)] \tag{3.69}$$

$$U\underline{\alpha}^z(P) = [(z(u \mid ((\exists\alpha)(\alpha \in U\underline{\alpha}_f(P))\ (u \in Uz(Q\ \alpha)] \tag{3.70}$$

$$F\underline{\alpha}_f(P) = (\alpha \mid (\alpha \notin T\underline{\alpha}_f(P))(\alpha \notin U\underline{\alpha}_f(P)] \tag{3.71}$$

$$F\underline{\beta}_f(P) = F\underline{\alpha}_f(P) \tag{3.72}$$

Definition 3.10: Residue of an Existentially Qunatified Expression

$$Fz^i(P\ \alpha) = Fz(Q_i\ \alpha) \tag{3.73}$$

and for ξ = U and T

$$\xi z^i(P\ \alpha) = (\xi z(Q\ \alpha) \cap \xi z(Q_i\ \alpha) \tag{3.74}$$

$$\xi\underline{\beta}_f^i(P) = ([\alpha\ (z\ z^i(P\ \alpha))] \mid (\alpha \in \xi\underline{\alpha}_f(P))(\xi z^i(P\ \alpha) \neq \lambda)] \tag{3.75}$$

$$\xi R[\underline{\alpha}_f]P = ((\exists z) \bigwedge_{i=1}^{k} (\xi R[\xi\underline{\beta}_f^i(P)]Q_i)) \tag{3.76}$$

Again, if $\xi\underline{\beta}_f^i(P) = \lambda$ then the corresponding Q_i will drop off the residue expression.

Definition 3.11: Partitions of Existentially Quantified Expressions

$$Q = ((\exists z)Q_1 Q_2 \ldots Q_k),\ k \geq 1,$$

$$UF_T\underline{\alpha}_f^i(Q) = (UF\underline{\alpha}_f(Q) \cap T\underline{\alpha}_f(Q_i)) \ldots \tag{3.77}$$

$$F_U\underline{\alpha}_f^i(Q) = (F\underline{\alpha}_f(Q) \cap U\underline{\alpha}_f(Q_i)) \tag{3.78}$$

$$UF_T\underline{\beta}_f^i(Q) = ([\alpha\ Tz(Q_i\ \alpha)] \mid (\alpha \in UF_T\underline{\alpha}_f^i(Q))(Tz(Q_i\ \alpha) \neq \lambda)) \tag{3.79}$$

$$F_U\underline{\beta}_f^i(Q) = ([\alpha\ Uz(Q_i\ \alpha)] \mid (\alpha \in F_U\underline{\alpha}_f^i(Q))(Uz(Q_i\ \alpha) \neq \lambda)] \tag{3.80}$$

$$\Pi T[\underline{\alpha}_f]Q = ((\exists z) \bigwedge_{i=1}^{k} (TR[UF_T\underline{\beta}_f^i(Q)]Q_i)) \tag{3.81}$$

$$\Pi U[\underline{\alpha}_f]Q = ((\exists z) \bigwedge_{i=1}^{k} (UR[F_U\underline{\beta}_f^i(Q)]Q_i)) \tag{3.82}$$

3.4: Computation of Residues and Partitions: An Overview

As presented above, the residue and partition computations require for each P, $\xi_{\underline{\beta}_f}(P)$, to be known. Since P is in mini-scope form, this would ultimately reduce to knowing $\xi_{\underline{\beta}_f}$ for the propositions contained by the various miniscope subexpressions of P. We shall review below the evaluation algorithm implied by the definitions given in the previous section, and briefly outline the method for the computation of $\xi_{\underline{\beta}_f}(M)$ for a proposition, M.

3.4.1. The Evaluation Algorithm

3.4.1.1. Single Universal Quantifier

$P = ((\forall z)M)$, $M = (M_1 \vee M_2 \vee \ldots \vee M_k)$, $k \geq 1$,

Each M_i is a propositional expression, which will contain z as one of its variables. We have the following solutions of M:

(M1): $\xi z(M_i\ \alpha) = (u \mid (\varphi[\alpha][(z\ u)]M_i = \varphi\xi))$

(M2): $Tz(M\ \alpha) = \bigcup_{i=1}^{k} Tz(M_i\ \alpha)$

(M3): $Uz(M\ \alpha) = (\bigcup_{i=1}^{k} Uz(M_i\ \alpha)) - Tz(M\ \alpha)$

(M4): $Fz(M\ \alpha) = \bigcap_{i=1}^{k} Fz(M_i\ \alpha)$

(M5): $\xi\underline{\alpha}_f(M) = (\alpha \mid (\xi z(M\ \alpha) \neq \lambda))$

(M6): $UF\underline{\alpha}_f(M) = (U\underline{\alpha}_f(M) \cup F\underline{\alpha}_f(M))$

(M7): $UF_T\underline{\alpha}_f(M) = (UF\underline{\alpha}_f(M) \cap T\underline{\alpha}_f(M))$

(M8): $F_U\underline{\alpha}_f(M) = (F\underline{\alpha}_f(M) \cap U\underline{\alpha}_f(M))$

(M9): $\xi\underline{\alpha}_s(M) = [(z(u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha}_f(M))$
$(u \in \xi z(M\ \alpha)]$

(M10): $UF_T\underline{\alpha}_s(M) = [(z(u \mid ((\exists\alpha)(\alpha \in UF_T\underline{\alpha}_f(M))$
$(u \in Tz(M\ \alpha)]$

(M11): $F_U\underline{\alpha}_s(M) = [(z(u \mid ((\exists\alpha)(\alpha \in F_U\underline{\alpha}_f(M))$
$(u \in Uz(M\ \alpha)]$

The above solutions may be used to obtain the ξ-solutions of P, as defined below:

(P1): $T\underline{\alpha_f}(P) = (\alpha \mid (Tz(M\ \alpha) = S_P(z))) \subseteq T\underline{\alpha_f}(M)$

where $S_P(z)$ is the total range of z in P.

(P2): $U\underline{\alpha_f}(P) = (\alpha \mid (\alpha \in U\underline{\alpha_f}(M))(Fz(M\ \alpha) = \lambda)) \subseteq U\underline{\alpha_f}(M)$

(P3): $F\underline{\alpha_f}(P) = F\underline{\alpha_f}(M)$

(P4): $UF_T\underline{\alpha_f}(P) = UF_T\underline{\alpha_f}(M)$

(P5): $F_U\underline{\alpha_f}(P) = F_U\underline{\alpha_f}(M)$

(P6): $\xi\underline{\alpha_s}(P) = [(z\ (u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha_f}(P))$

$(u \in \xi z(M\ \alpha)]$

(P7): $UF_T\underline{\alpha_s}(P) = UF_T\underline{\alpha_s}(M)$

(P8): $F_U\underline{\alpha_s}(P) = F_U\underline{\alpha_s}(M)$

We may write

(P9): $\xi\underline{\beta_f}(P) = [\xi\underline{\alpha_f}(P)][\xi\underline{\alpha_s}(P)]$

with the interpretation

$(\forall\alpha)(\alpha \in \xi\underline{\alpha_f}(P)) \Rightarrow ((\exists\delta)(\delta \in \xi\underline{\alpha_s}(P))(\varphi[\alpha][\delta]P = \varphi\xi))$

We have the following identities:

(i) $T\underline{\alpha}_f(P) \cup UF_T\underline{\alpha}_f(P) = T\underline{\alpha}_f(M)$

(ii) $U\underline{\alpha}_f(P) \cup F_U\underline{\alpha}_f(P) = U\underline{\alpha}_f(M)$

(iii) $T\underline{\alpha}_s(P) = S_T(z)$

To compute the residues and partitions of P, the solutions of P are redistributed among M_i, for $1 \leqslant i \leqslant k$, as follows:

(P10): $\xi\underline{\alpha}_f^i(P) = (\xi\underline{\alpha}_f(P) \cap \xi\underline{\alpha}_f(M_i))$

(P11): $UF_T\underline{\alpha}_f^i(P) = (UF_T\underline{\alpha}_f(P) \cap T\underline{\alpha}_f(M_i))$

(P12): $F_U\underline{\alpha}_f^i(P) = (F_U\underline{\alpha}_f(P) \cap U\underline{\alpha}_f(M_i))$

(P13): $\xi\underline{\alpha}_s^i(P) = [(z\ (u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha}_f^i(P))$

$(u \in \xi z(M_i\ \alpha)]$

Similarly $UF_T\underline{\alpha}_s^i(P)$ and $F_U\underline{\alpha}_s^i(P)$ are also defined. Using these redistributed solutions, the residues and partitions of P may be computed as shown below:

(P14): $\xi R[\xi\underline{\alpha}_f(P)]P = ((\forall z) \bigvee_{k=1}^{k} (\xi R[\xi\underline{\alpha}_f^i(P)][\xi\underline{\alpha}_s^i(P)]M_i))$

(P15): $\Pi T[\underline{\alpha}_f(P)]M_i = (TR[T\underline{\alpha}_f^i(P)][T\underline{\alpha}_s^i(P)]M_i \wedge$

$TR[UF_T\underline{\alpha}_f^i(P)][UF_T\underline{\alpha}_s^i(P)]M_i).$

(P16): $\Pi T[\underline{\alpha}_f]P = (\bigvee_{i=1}^{k} ((\exists z)\ \Pi T[\underline{\alpha}_f]M_i))$

(P17): $\Pi U[\underline{\alpha}_f]M_i = (UR[U\underline{\alpha}_f^i(P)][U\underline{\alpha}_s^i(P)]M_i \wedge$

$UR[F_U\underline{\alpha}_f^i(P)][F_U\underline{\alpha}_s^i(P)]M_i)$

$$(P18)\quad \Pi U[\underline{\alpha}_f]M = \left(\bigvee_{i=1}^{k} ((\exists z)\, \Pi U[\underline{\alpha}_f(P)]M_i)\right)$$

If $\xi\underline{\alpha}_f^i(P) = \lambda$ for a given i, then the corresponding M_i will drop off the residue expression. So also, if $UF_T\underline{\alpha}_f^i(P)$ or $F_U\underline{\alpha}_f^i(P)$ is λ then the corresponding M_i will drop off the the partition expressions, ΠT and ΠU, respectively. Let us now consider the solutions for expressions with a single existential quantifier.

3.4.1.2. Single Existential Quantifier

$Q = ((\exists z)N)$, $N = (N_1\ N_2\ \ldots\ N_k)$, $k \geq 1$,

Each N_i is a propositional expression with z as one of its free variables. We have then the following solutions:

$$(N1):\quad \xi z(N_i\ \alpha) = (u \mid (\varphi[\alpha][(z\ u)]N_i = \varphi\xi))$$

$$(N2):\quad Tz(N\ \alpha) = \bigcap_{i=1}^{k} Tz(N_i\ \alpha)$$

$$(N3):\quad Uz(N\ \alpha) = \left(\bigcup_{i=1}^{k} Uz(N_i\ \alpha)\right) - Fz(N\ \alpha)$$

$$(N4):\quad Fz(N\ \alpha) = \bigcup_{i=1}^{k} Fz(N_i\ \alpha)$$

$$(N5):\quad \xi\underline{\alpha}_f(N) = (\alpha \mid (\xi z(N\ \alpha) \neq \lambda))$$

$$(N6):\quad \xi\underline{\alpha}_s(N) = [(z\ (u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha}_f(N))\ (u \in \xi z(N\ \alpha)]$$

Using these, the following solutions for Q may be obtained:

(Q1): $T\underline{\alpha_f}(Q) = T\underline{\alpha_f}(N)$

(Q2): $U\underline{\alpha_f}(Q) = (\alpha \mid (Tz(N\ \alpha) = \lambda)(\alpha \in U\underline{\alpha_f}(N)))$

(Q3): $F\underline{\alpha_f}(Q) = (\alpha \mid Fz(N\ \alpha) = S_Q(z)))$

(Q4): $\xi\underline{\alpha_s}(Q) = [z(u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha_f}(Q))$
$(u \in \xi z(N\ \alpha)]$

(Q5): $UF_T\underline{\alpha_f}(Q) = F_U\underline{\alpha_f}(Q) = UF_T\underline{\alpha_s}(Q) = F_U\underline{\alpha_s}(Q) = \lambda$

However, we will have the following solutions for each N_i:

(Q6): $UF_T\underline{\alpha_f^i}(Q) = (UF\underline{\alpha_f}(Q) \cap T\underline{\alpha_f}(N_i))$

where $UF\underline{\alpha_f}(Q) = (U\underline{\alpha_f}(Q) \cup F\underline{\alpha_f}(Q))$

(Q7): $F_U\underline{\alpha_f^i}(Q) = (F\underline{\alpha_f}(Q) \cap U\underline{\alpha_f}(N_i))$

(Q8): $UF_T\underline{\alpha_s^i}(Q) = [(z\ (u \mid ((\exists\alpha)(\alpha \in UF\underline{\alpha_f^i}(Q))$
$(u \in Tz(N_i\ \alpha)$

(Q9): $F_U\underline{\alpha_s^i}(Q) = (z\ (u \mid ((\exists\alpha)(\alpha \in F_U\underline{\alpha_f^i}(Q))$
$(u \in Uz(N_i\ \alpha)]$

(Q10): $\xi\underline{\alpha_f^i}(Q) = \xi\underline{\alpha_f}(Q)$

(Q11): $\xi\underline{\alpha_s^i}(Q) = \xi\underline{\alpha_s}(Q)$

Using these, the residues and partitions may be written as follows:

(Q12): $\xi R[\xi\underline{\alpha_f}(Q)]Q = ((\exists z) \bigwedge_{k=1}^{k} (\xi R[\xi\underline{\alpha_f}(Q)][\xi\underline{\alpha_s}(Q)]N_i))$

(Q13): $\Pi T[\underline{\alpha_f}]N_i = (TR[UF_T\underline{\alpha_f^i}(Q)][UF_T\underline{\alpha_s^i}(Q)]N_i)$

(Q14): $\Pi T[\underline{\alpha}_f]Q = ((\exists z) \overset{k}{\underset{i=1}{\wedge}} (\Pi T[\underline{\alpha}_f]N_i))$.

(Q15): $\Pi U[\underline{\alpha}_f]N_i = ((UR[F_U\underline{\alpha}_f^i(Q)][F_U\underline{\alpha}_s^i(Q)]N_i)$

(Q16): $\Pi U[\underline{\alpha}_f]Q = ((\exists z) \overset{k}{\underset{i=1}{\wedge}} (UR[\underline{\alpha}_f]N_i))$

If P in 3.4.1.1 and Q above have several quantifiers, then the redistribution of the bindings to M_i (N_i) will take place only after all the quantifier checks are completed. The organization of the binding for this case is presented in the next section, below.

3.4.1.3: General Predicate Expressions

Case (a): $P = ((\forall z)Q)$, $Q = (Q_1 V Q_2 V \ldots V Q_k)$, $k \geq 1$, or

Case (b): $Q = ((\exists z)P)$, $P = P_1 P_2 \ldots P_k$, $k \geq 1$,

where each P_i (and Q_i) is itself a quantified expression with z as one of its free variables. After completing all the quantifier checks for each P_i (Q_i), we will have the solution, in the following form:

(Z1): $\xi\underline{\delta}_f(Z_i) = [\xi\underline{\beta}_f(Z_i)][\xi\underline{\alpha}_s(Z_i)]$,

where Z_i is a P_i or a Q_i, for $1 \leq i \leq k$. $\xi\underline{\alpha}_s(Z_i)$ will contain the solutions for the quantified variables, that are local to Z_i. It may be noticed that this form is the same one, shown in (P9). Using $\xi\underline{\beta}_f(Z_i)$, the solution for case (a) and (b) above may now be obtained.

For Case (a) one should use formulas (M1) through (M9) and (P1) through (P18), after doing the following substitutions:

Replace (M1) by,

$$(M1'): \xi z(Q_i\ \alpha) = (u \mid ([\alpha\ (z\ u)] \in \xi_{\underline{\beta}_f}(Q_i)))$$

and substitute throughout Q for M and Q_i for M_i. In formulas (P13) through (P18) the solutions $\xi_{\underline{\alpha}_s}^{i}(P)$ will contain not only the bindings specified for z, but also the bindings $\xi_{\underline{\alpha}_s}(Q_i)$ shown in (Z1) above, for $Z_i = Q_i$. That is, $\underline{\alpha}_s^i$ solutions will specify binding ranges not only for z, but also for all the local variables of Q_i. To indicate this, $\xi_{\underline{\alpha}_s}^{i}(P)$ may be respecified as follows:

$$\xi_{\underline{\alpha}_s}^{i}(P) = (\delta_u \mid ((\exists\alpha)(\alpha \in \xi_{\underline{\alpha}_f}^{i}(P)) (\varphi[\alpha][\delta_u]Q_i = \varphi\xi))$$

where δ_u specifies the bindings for z and all the other local variables of Q_i. Similar considerations apply also for other $\underline{\alpha}_s^i$ solutions of P.

Similarly, for case (b) the solutions may be obtained, again by using $\xi_{\underline{\beta}_f}(Z_i)$ for $Z_i = P_i$, and the formulas (N1) through (N6) and (Q1) through (Q16), after the following changes:

Replace (N1) by

$$(N1'): \xi z(P_i\ \alpha) = (u \mid ([\alpha\ (z\ u)] \in \xi_{\underline{\beta}_f}(P_i)))$$

and substitute throughout, P for N and P_i for N_i.

As in case (a), $\xi\underline{\alpha}^i_s(Q)$ will be specified by

$$\xi\underline{\alpha}^i_s(Q) = (\delta_u \mid ((\exists\alpha)(\alpha \in \xi\underline{\alpha}^i_f(Q)) \\ (\varphi[\alpha][\delta_u]P_i = \varphi\xi)$$

where δ_u specifies the binding ranges for z and all other local variables of P_i. Again, similar considerations apply also to other α^i_s solutions of Q.

The evaluation algorithm implied by the above definitions may now be summarized as consisting of three steps:

Step 1: Finding solutions for propositions like M,N,M_i and N_i, that occur within mini-scope expressions.

Step 2: Doing quantification check for the mini-scope expressions, proceeding outwards.

Step 3: After completing all quantification checks, redistributing the solutions of the mini-scope expressions, among their propositional components. These redistributed bindings are used for calculating the appropriate residues and partitions of P. These residues and partitions will be computed, of course, only after evaluating an entire predicate expression. In the case of CC's, this will amount to the evaluation of the set predicate, SP, of the CC.

The algorithm used for step (1) above, is briefly outlined below:

3.4.2. Computation of Solutions for Propositional expressions

For each elementary form $(p\ x_1 \ldots x_n)$ or $\sim(p\ x_1 \ldots x_n)$, for $n \geq 2$, one may easily obtain from the model space the solutions $x_i(P\ \alpha)$, for given α and i, $1 \leq i \leq n$. The availability of the anchor, @, in CC makes it possible to assign to each elementary form in a CC a free range α, and thus determine for each propositional form in a CC, its associated free range and solutions. The scheme for doing this is briefly outlined below.

The elementary forms in a CC are ordered in a level tree, as shown in figure 3. All binary relational forms in which @ occurs appear at the top of the tree. At level 1 we have the anchor @. At level 2 we have all variables, x, for which forms "(@ r x)" or "(x r @)" occur in the CC. At the third level, one has all the variables, y, for which "(x r y)" or "(y r x)" occur in the CC. Continuing in this manner, the various levels of the tree are filled. It is possible that a given variable has more than one arc impinging on it. Also, variables at the same level may have arcs between them.

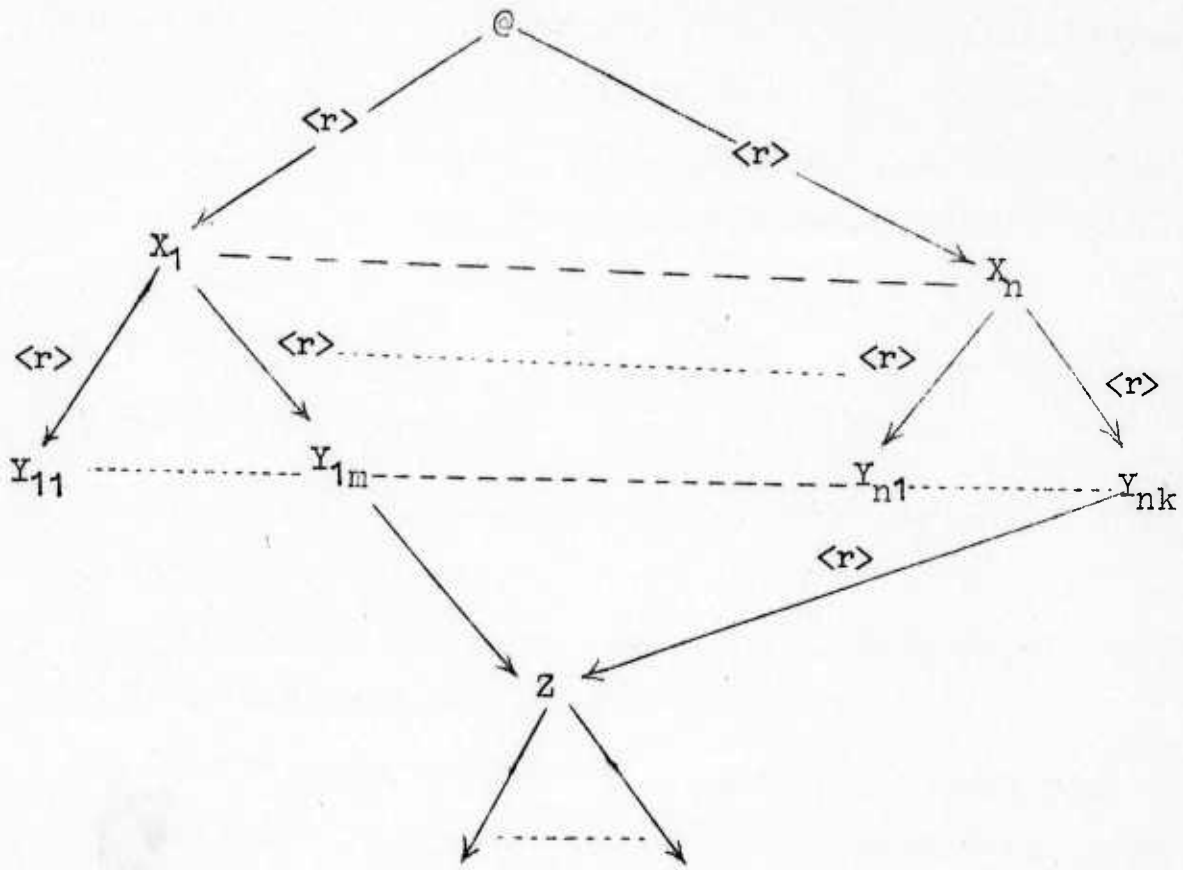

Fig. 3: The Level Tree of a CC

All the relations with "@" are evaluated first. This will cause value ranges to be assigned to the variables at the second level. These bindings may then be used to evaluate the relations at the next level, thereby fixing the ranges for the variables in the next level below. For each variable, its associated evaluation sequences will be repeated until the maximal, in case of disjunctions, or mimimal, in case of conjunctions, solution to the variable is obtained. The tuples and function forms are evaluated last.

The bindings obtained for the variables have to be organized to serve two purposes:

(a) For use in the quantifier checks, discussed in the previous section, and

(b) For recognition of conjunction or disjunction frames within which the bindings of a variable occur.

To serve the purpose (a), the variables in a mini-scope expression are ordered in the order of their quantification: The variable of the innermost quantifier appearing last. If a variable, x_i, occurs in an ordering, $x_1 \ldots x_i \ldots x_n$, but does not occur in a miniscope form, P, with which the ordering is associated, then by definition the scope of x_i in P is universal.

To serve the purpose (b), for each conjunction (disjunction)

in a CC, a conjunctive (disjunctive) frame is created. This frame will contain the variables that occur in the conjunction (disjunction). Each variable will contain pointers to the quantifier associated with it, to the relations in the CC in which it occurs, and to the next variable in the ordering outlined in the previous paragraph. Each variable will also contain slots for storing the partitions of its range, induced by the CC. We shall associate with this frame, also the procedures necessary to evaluate the relations in the frame, and update the bindings. Any time the bindings associated with a variable is changed, the relevant relations of the variable will be re-evaluated. In a conjunctive frame the minimal solution of a variable will be obtained by successive intersections. In a disjunctive frame, the maximal solution is obtained by successive unions.

The frame data structures created for the variables in a CC would thus depend on the structure of the CC. The collection of all such frames, created for a CC, is called the CC-associative net, CCA-net. The CCA-net, the evaluation order for relations, the updating rules for the CCA-net, and the procedures for quantification checks and residue representations may all be compiled from the CC-definition. The details of this compilation process and the complexity of the procedures involved will be discussed in a future paper.

A fully instantiated CCA-net will represent the complete solution of a CC. All the information of the residues and partitions of a CC may be obtained from this net. The representations of residues and partitions of a CC-evaluation constitute a summary of the information in the CCA-net, in a form suitable for communication in the language of the CC.

3.4.3: Comments on the evaluation process

To have an effective model space, and do commensense reasoning, it is essential that CC-evaluations be efficient. In MDS organization this is facilitated by several features. The most basic of these is the way the relation values are stored in the model space, as discussed in Appendix II. This enables one to fetch easily the partitions induced by a relation on the range of a variable. The second feature is parallelism: One may incorporate parallelism in a CC evaluation process at various levels. At the level of relations, the elementary forms at the same stage of a level tree may all be evaluated in parallel. At the level of miniscope expressions, the conjunctive and disjunctive frames associated with different miniscope expressions of a CC may be evaluated in parallel. The casting of a CC in mini-scope form reduces the height of nested quantifications in a CC. This simplifies the structure and processing of the conjunctive and disjunctive frames. The quantification checks associated with miniscope forms may all be also executed in parallel.

Finally, the residue and partition extraction for the mini-scope forms may be done in parallel. This evaluation process avoids back tracking. This is partly because it attempts always to obtain the complete solution, and partly because the relation evaluation sequence can be ordered, as per the level tree. Also, for every binary relation its complete solution may be obtained easily from the model space.

The possibility of compiling the CCA-net and its associated processors further enhances efficiency. One may, in fact, conceiv of machine organizations in which the evaluation algorithm for a CC may be micro-programmed, taking full advantage of the parallel processing possibilities. We have presented here only the bare outlines of the evaluation process.

3.5. Commonsense Reasoning and Problem Solving

3.5.1: The Basic Theorem

Let $Q([\underline{\alpha}_f]P\ M_i)$ denote the truth value of $[\underline{\alpha}_f]P$ in model state, M_i. So also, let $\xi R([\underline{\alpha}_f]P\ M_i)$ and $\Pi\xi([\underline{\alpha}_f]P\ M_i)$ denote the ξ-residue and ξ-partition, respectively, in model state, M_i. Let $\Delta\underline{\alpha}_f$ be the change in $\underline{\alpha}_f$ in a new model state, M_j, such that the new free range is $[\underline{\alpha}_f + \Delta\underline{\alpha}_f]$. For each variable in $\underline{\alpha}_f$, the change $\Delta\underline{\alpha}_f$, may specify deletions and/or additions to its range of values.

Let

$$\Delta\underline{\alpha}_f(\xi R([\underline{\alpha}_f]P\ M_i)) \text{ and } \Delta\underline{\alpha}_f(\Pi\xi([\underline{\alpha}_f]P\ M_i))$$

denote the residues and partitions with the changes in $\Delta\underline{\alpha}_f$ incorporated in them. The theorems in this section pertain to the inferences that may be drawn on the truth values of

$$\varphi([\underline{\alpha}_f + \Delta\underline{\alpha}_f]P\ M_j)$$

based on the truth values of

$$\varphi(\Delta\underline{\alpha}_f(\xi R([\underline{\alpha}_f]P\ M_i))M_j) \text{ and}$$

$$\varphi(\Delta\underline{\alpha}_f(\Pi\xi([\underline{\alpha}_f]P\ M_i))\ M_j).$$

The theorems are presented below. Their applications are discussed in the next section.

Theorem 1: Updating/Learning Theorem

For ξ = T and F

$$[((\varphi(\Delta\underline{\alpha}_f(\xi R([\underline{\alpha}_f]P\ M_i))\ M_j) = \varphi\xi) \Rightarrow (\varphi([\underline{\alpha}_f + \Delta\underline{\alpha}_f]P\ M_j) = \varphi\xi)] \quad (3.83)$$

Proof is by induction on the structure of P: P is a p or a $\sim$p, or P is $(P_1 * P_2 * \ldots * P_k)$, $k \geq 1$, and $* = \wedge$ or V, or P is $((\forall z)(P_1 V P_2 V \ldots V P_k))$ or $((\exists z)P_1 P_2 \ldots P_k)$,

Theorem 2: Means-end Analysis Theorems

(A): $$[(\varphi(\Delta\underline{\alpha}_f((UR([\underline{\alpha}_f]P\ M_i)) \wedge (\Pi T([\underline{\alpha}_f]P\ M_i))))\ M_j) = (\varphi([\underline{\alpha}_f + \Delta\underline{\alpha}_f]P\ M_j) \quad (3.84)$$

(B): $$\varphi((\Delta\underline{\alpha}_f((FR([\underline{\alpha}_f]P\ M_i))(\Pi U([\underline{\alpha}_f]P\ M_i)) \wedge (\Pi T([\underline{\alpha}_f]P\ M_i))))\ M_j) = (\varphi([\underline{\alpha}_f + \Delta\underline{\alpha}_f]P\ M_j)]. \quad (3.85)$$

Again, the proof is by induction on the structure of P.

3.5.2. Updating and Learning Theorem

Our discussion in this section pertain mostly to the T and F residues. All statements made here also apply to $UR([\underline{\alpha}_f]P\ M_i)$, if it exists, if the residue check included also $\Pi T([\underline{\alpha}_f]P\ M_i)$: that is the residue checks evaluated the left side of (3.84),

instead of just the residues.

3.5.2.1: Basis for Simplification of Consistency checks during Updating

When an anchor $[@_X\ r]$ is updated there will usually be a set of invocation anchors, at which the relation values are changed. Let this set be

$$B = \{(@_X\ r\ y_1),\ (@_{X_1}\ r_1\ y_2),\ldots,\ (@_{X_n}\ r_n\ y_n)\} \qquad (3.86)$$

For every relation in B, the new relation value, y_i, $1 \leq i \leq n$, is mandatory.

For an invocation anchor $[@_X\ r]$ implied by B -- i.e. $(@_X\ r\ y)$ is in B -- if

$$\varphi(\Delta\alpha_f(FR(CC[X\ r](@_X)\ M_i))\ M_j) = NIL \ldots \qquad (3.87)$$

then there would exist a subset, S_1, of the new values of $(@_X\ r)$, for which the F-residue at $[@_X\ r]$ evaluates to NIL. Then, by Theorem 1, it follows that the values in S_1 may only be assigned as the complement solution of $(@_X\ r)$, i.e. $S_1 \subseteq (z \mid \sim(@_X\ r\ z))$.

Similarly, if for a subset S_2 of the new values of $(@_X\ r)$, the true-residue at $[@_X\ r]$ evaluates to T, i.e.

$$\varphi(\Delta\alpha_f\ (TR(CC[X\ r](@_X)\ M))\ M_j) = T, \qquad (3.88)$$

then the new values in S_2 may be assigned as the true values of

$(@_X\ r)$, provided that none of values in S_2 cause a contradiction in the interaction checks associated with $[@_X\ r]$.

If for ξ = T or F,

$$\varphi(\Delta\underline{\alpha}_f(\xi R(CC[X\ r](@_X)\ M_i))\ M_j) \neq \varphi\xi \qquad (3.89)$$

then the entire CC[X r] should be re-evaluated at $@_X$.

While evaluating CC[X r]$(@_X)$, one need find new solutions, $\xi\underline{\beta}_f(N)$, for the propositions in the CC, only if they do not appear in the ξ-residues of the CC for ξ = T and F. Also, in finding $\xi\underline{\beta}_f(N)$, the re-evaluation may be done only for the changes specified by $\Delta\underline{\alpha}_f$ and B. Thus, CC re-evaluation using residues may always be done more economically than direct CC evaluations.

In the case of interaction checks, the free range, $\underline{\alpha}_f$, would remain unchanged, for every $[@_Y\ t]$ that depends on an $[@_X\ r]$ in B. But, some of the relations appearing in $CC[@_Y\ t]$ or its ξ-residues would have changed their values. To check for consistency, the ξ-residues at $[@_Y\ t]$ may be re-evaluated with respect to these changes. The contradiction check may be done as per rules

$$[\varphi(TR(CC[Y't](@_Y\ s)\ M_i))\ M_j) = T) \Rightarrow$$

$$(@_Y\ t\ s)], \qquad (3.90)$$

and

$$[\varphi(FR(CC[Y\ t](@_Y\ s)\ M_i))\ M_j) = NIL \Rightarrow \sim(@_Y\ t\ s) \qquad (3.91)$$

Again, as in the previous case, the entire CC, $CC[Y\ t](@_Y)$ itself would be evaluated only if (3.89) is true for $CC[Y\ t](@_Y)$.

Thus, by direct application of Theorem 1, one may simplify consistency checks during an updating process. Other applications of this theorem arise in goal directed problem solving. They set the basis in MDS for learning. This is discussed below

3.5.2.2: The Basis for Learning

Goals are stated in MDS in the form

$$(G\ x\ y...z) = (\ \langle\text{binding-conditions}\rangle\ (\text{GOAL}\ \langle\text{goal-conditions}\rangle)), \qquad (3.92)$$

where x,y,...,z are the free variables of <binding-conditions>. The binding conditions will specify the initial conditions for the goal, and the objects that may be used to achieve the goal. The <goal-condition> will always be a conjunction of elementary forms. The above goal statement may be interpreted as follows:

"The <goal-conditions> are to be satisfied for the objects satisfying the <binding-conditions>, for given ranges of the free variables x,y,...,z."

Each free variable will, of course, have an associated scope.

We shall refer to statements like (3.92) as the dimensions of goals. The dimension of a goal specifies the nature of the goal. Thus, for example, the dimension.

((PEOPLE X)(PLACE P Q)(P location of X))
(GOAL(Q locationof X)) (3.93)

is the dimension of a goal called, say (MOVE-PEOPLE P Q X). It describes the nature of this action. This goal may have associated with it a body, that specifies the action for achieving the goal, namely

(XR Q X) = ((SOME VEHICLE V)
(ASSERT (V holding X))
(ASSERT (V location Q))
(ASSERT~(V holding X) (3.94)

In general, a goal like MOVE-PEOPLE, may have several transformation rules associated with it. Each transformation rule, XR, will have the form:

(XR x y...z) = (<binding-conditions> <actions>), (3.95)

where the <actions> may be subgoal statements or ASSERT,DELETE or CREATE statements. An XR is said to match a goal, G1, if there is a subset of changes that the XR may cause, which matches with a subset of the conjuncts in the goal condition of G1, and the solutions of the binding conditions of G1 do not contradict the

binding conditions of XR. The true residues of these binding conditions of G1 and XR, will then characterize the context of invocation of the XR called the dimension of XR invocation, or the invocation dimensions of the XR.

An invoked XR may be tried only if the preconditions associated with the invocation dimension are satisfied, for the objects involved in the XR. These pre-condition statements will have the form,

```
(CANDO (<binding conditions> <actions>)
       (<conditions> (TRY <action>)
       (<conditions> (TRY <actions>)...
       ...
       (<conditions> (TRY <actions>)]                    (3.96)
```

A CANDO-statement is said to match an invocation dimension of an XR if the "(<binding-condition> <actions>)" of the CANDO-statement match with the invocation dimension of the XR. A CANDO-statement is said to be satisfied if none of the <conditions> in the statement are contradicted, or for every condition that is contradicted its associated TRY-statement was executed successfully. Again, the satisfaction or non-satisfaction of a CANDO-statement would return residues that explain the reasons for the outcome. We shall refer to these reasons as the <precondition checks>.

Finally, the execution of the <actions> in an XR would result in a collection of residues that explain the reasons for

the success, failure or conditional success of the actions. In the invocation context of an XR, one might have had several choices for the relations and objects on which the <actions> might be taken. For the choices made, the residues returned by the actions may be used to characterize the choices, in the form of positive and negative focus lists, associated with the XR. Thus, for example, if the assertion, (ASSERT (V holding X)), in (3.94) failed for the reason, ~(X #: ≤ : capacityof V), then this piece of information may be summarized at the XR in the form of a negative focus list element,

NFL[V holding][XR] =

([V X]| ~(X #: ≤ : capacityof V)) (3.97)

On a later invocation of the XR, the same assertion will not be tried for object combinations, [V X], that satisfy the above focus list element.

Similarly, the relations and objects for which actions succeeded may be characterized by positive focus lists associated with an XR. The focus list predicates would consist of T and/or U residues returned by the ASSIMILATOR, for the relations and objects involved. On a future invocation of the same action the relations and objects satisfying the positive focus list will be preferentially chosen.

XR's in MDS may also a have post-conditions associated with them. These have the same form as the CANDO-statement. But the

word "IFDONE" is used instead of "CANDO". Evaluation of these post-conditions also would return residues. We shall refer to these as the <post-condition checks>.

The invocation and execution (also called the instantiation) of an XR would thus result in residues that characterize the instantiation, in terms of four components:

I[XR] = [<dimension> <pre-condition checks>
<focus list> <post condition checks>] (3.98)

The DESIGNER is responsible for gathering together the residues and associating them in the right manner with the XR's and goals, and thus maintain an updated problem solving state. A brief discussion of this process appears in Srinivasan [1973]. A more detailed discussion will appear in Srinivasan [1977c].

In (3.98) the <dimension> and <precondition checks> together specify the appropriateness of an XR in a given invocation. The <focus lists> have validity only in the given <dimension> and <precondition check> context. In another invocation of the XR, if the <dimension> and <precondition check> of the invocation match with those in (3.98) then the <focus lists> in (3.98) may be used to guide search according to the rules given below:

(i) Avoid choosing relations and object combinations that satisfy elements in the negative focus list of matching XR invocation.

(ii) Choose preferentially the objects and relations that satisfy the elements in the positive focus list of matching XR invocations.

(iii) If choices as per (i) and (ii) do not succeed then update focus lists and repeat (i) and (ii) for new untried combinations.

By theorem 1, the objects that satisfy the true-residues in the positive focus lists are likely to succeed again, and those that satisfy the false-residues in the negative focus lists are likely to fail again.

The focus lists associated with a goal dimension will depend on all the XR's executed to achieve the goal. Again, the DESIGNER is responsible for constructing the summarizing focus lists for a goal dimension, based on the focus-lists of the associated XR executions. As XR executions are repeatdd in different execution contexts the DESIGNER will acquire progressively more domain knowledge from the model space, in the form of residues. These residues, taken together and summarized as focus lists, would represent combinations of chunks of domain knowledge unique to a goal (problem) or task environment. The use of focus lists as per rules (i), (ii) and (iii) above sets the basis for learning in MDS. Through its interactions with the model space, the DESIGNER acquires for each problem, its own unique way of viewing the model space and using the domain knowledge. As mentioned

before, Theorem 1 sets the logical basis for this learning.

The focus lists are used in MDS for a variety of purposes. Two other uses of focus lists are discussed below.

3.5.2.3: Use of focus lists to guide intelligent conversation

The predicate expressions in the characterization (3.98) of I[XR], may be used selectively by a system to conduct a conversation with a user, concerning the nature, appropriateness and reasons for success/failure etc., of an XR execution. For a given goal, the residues associated with the goal may be large and quite numerous. To maintain an intelligent conversation there should then be some rules available for judicious selection of residues and predicates within the residues, which could be used for the conversation. The positive and negative focus lists may be used for doing this selection.

In explaining the reasons for the failure of an action the negative focus list is usually significant: The action failed because it contradicted some of the fixed relations and values. While explaining the reasons for the success of an action the positive focus list is usually significant: The action succeeded because some of the secondary changes in the positive focus list of an XR, succeeded.

Thus, while explaining the reasons for the success of the assertion, (ASSERT(V holding X)), the fact that
(X # : ≤:capacityof V)

was true, would not be of much interest. Here, the phrase [VEHICLE capacity] might belong to the negative focus list of the action. However, if the action failed, then the reason ~(X # : ≤ : capacityof V), becomes significant, and forms the basis for corrective reaction.

Similarly, suppose the action failed because-- ~(V location:locationof X). In this case [VEHICLE location] may belong to the positive focus list of the action. Thus, subsequent action might have been invoked to change (V location) to (X location). In this case, part of the reason for the success of the action, (V holding X), would be the reaction to the initial failure. The dimension of this reaction,namely:

[((PEOPLE X)(VEHICLE V)(PLACE P)
(X location P) ~(V location P))
(GOAL (V location P)], (3.99)

would thus be a legitimate part of the reasons for the success of the assertion.

3.5.2.4: Use of focus lists to guide Recognition

The transformation rules in MDS may be used in two ways. To plan and execute actions to reach a given gc ", or to recognize given sequences of actions as constituting a familiar goal. The recognition task may be posed to MDS as one of constructing the

dimensions that may be associated with a given sequence of actions. The central problem in this task is a classification problem: The actions are to be classified into two groups. Those that may be considered the principal actions, and those that are the "side-effects" that accompany the principal actions in the modelspace. Here, one may use the positive focus lists associated with the actions to find one or more minimal sets of principal actions, whose focus lists account for all the remaining actions in the sequence. The dimension possible for the action sequence may then be constructed in terms of the initial and final conditions associated with the principal actions. These dimensions may then be matched against known goals in the system, to complete the recognition process.

It is interesting to note here, that the focus lists are built up as a result of interactions with the model space. As the focus list repertoir becomes rich the recognition tasks would become simpler. Thus, availability of goal statements and XR's is not by itself sufficient, to do good recognition. Experience with the use of the goals and XR's in the model space is necessary. This illustrates another aspect of learning, in MDS.

Focus lists thus play a central role in problem solving in MDS. They provide a format for summarizing and using domain knowledge. The predicate expressions that characterize the elements of focus lists are obtained from the residue extraction process. The logical basis for the use of residues in this manner is provided by Theorem 1.

3.5.3. Basis For Theorem Proving

Suppose one wanted to prove the assertion

[(HUMAN h)(SOME PLACE p)
(h location p)] (3.98)

This asserts that for every HUMAN, h, there is a PLACE, p, such that (h location p) is true. In our model space, if there existed a HUMAN, h_o, such that there was no PLACE, p, for which (h_o location p) was true, then this would contradict the schema [S4], introduced in section 2.3.1,

[S4]: (HUMAN location PLACE)

Hence, (3.98) is true in our domain. We shall present below the formalization of this proof, as it would appear in MDS, and use it, to illustrate the basis for theorem proving in MDS. The TP-control structure in MDS, and its use of Gentzen's system of logic (see Kanger 1963 for a brief exposition of this logic, and [Beth 1959] for an interesting discussion), are outlined in Appendix III. The essentials of this logic necessary to understand the example in this section, are presented below.

3.5.3.1: Gentzen's System of logic and the calculus of sequents.

A **sequent** is of the form

$$(P_1)(P_2)\ldots(P_n) \rightarrow (Q_1)(Q_2)\ldots(Q_n): \qquad (3.99)$$

for $n,m \geq c$, where each P_i and Q_j is a predicate expression in miniscope form. The left side is interpreted as a conjunction, and the right side, as a disjunction. We will use symbols, S, S_1, etc., to denote an arbitrary sequent.

A TP-state, TP_i, will consist of a set of such sequents. We shall use ";" to separate the sequents in a TP-state.

The calculus of sequents, proposed by Gentzen, provides a systematic way of expanding the quantified expressions in a sequent, to generate sequents of the form,

$$(p_1)(p_2) \ldots (p_n)\ S_1 \rightarrow S_2(q_1)(q_2) \ldots (q_m); \qquad (3.100)$$

where each p_i and q_j is a literal (an elementary form), without negation. The expansion of quantifiers in the sequents of a TP-state would result in the generation of two kinds of variables:

Eigen Variables: Each eigen variable denotes a unique and distinct instantiated object.

Eigen Terms: Each eigen term is an unbound variable with an associated range. These are the free variables in the expressions appearing in sequents. The range of an eigen term may consist only of the eigen variables generated by the calculus.

In a TP-state, TP_i let, $EV(TP_i)$, denote the collection of all eigen variables in TP_i, and $ET(TP_i)$, the eigen terms in TP_i.

The rules for the generation of these variables are given below. Each rule is of the form

$$\frac{S_3 \rightarrow S_4;}{S_1 \rightarrow S_2;}$$

signifying that a sequent of the form "$S_1 \rightarrow S_2$" in a TP-state, may be replaced by another of the form "$S_3 \rightarrow S_4$;".

Generalization Rules:

These generate eigen terms, z.

$$(\rightarrow \exists): \quad \frac{S_1 \rightarrow S_2((\exists x)(Qx)S_3([(x\ z)]Qx)}{S_1 \rightarrow S_2((\exists x)(Qx)S_3} \tag{3.101}$$

$$(\forall \rightarrow): \quad \frac{([(x\ z)]Px)S_1((\forall x)Px)S_1 \rightarrow S_3}{S_1((\forall x)Px)S_2 \rightarrow S_3} \tag{3.102}$$

where z is a new variable that does not occur in the TP state, TP_i, in which the sequents in the dinominator appear. The range of z is, $EV(TP_i)$.

Instantiation Rules:

These generate eigen variables, a.

$$(\rightarrow \forall): \quad \frac{S_1 \rightarrow S_2((\forall x)Qx)S_3([(x\ a)]Qx)}{S_1 \rightarrow S_2((\forall x)Qx)S_3} \tag{3.103}$$

and

$$(\exists \rightarrow): \quad \frac{([(x\ a)]Px)S_1((\exists\ x)Px)S_1 \rightarrow S_3}{S_1((\exists\ x)Px)S_1 \rightarrow S_3} \qquad (3.104)$$

where "a" does not occur in the TP state, TP_i, in which the sequents in the dinominator appear. "a" denotes an unique, newly instantiated object. The adaptation of these expansion rules, and other propositional rules of Gentzen to MDS is discussed in Appendix III.

3.5.3.2: A proof example

We shall state the theorem to be proven, namely statement (3.98), as a sequent:

(M) → ((HUMAN h)(SOME PLACE p)
(h location p)); (3.105)

where :(M) is a conjunction of all true assertions in the model space. Applying the rule (→ ∀) (the rule (3.103)), the above sequent may be transformed to

(M) → ((SOME PLACE p)(h_o location p)); (3.106)

where h_o is an eigen variable created by issuing the command: (CREATE HUMAN h_o). Initially, all the relations of h_o will have the value "?", assigned to them. Thus, the model space will have the following:

[(HUMAN instance (h_o ?))
(h_o type ?)(h_o candrive (?))
(h_o compatiblewith (?))(h_o holding (?))
(h_o location ?)].

The rule (→ ∃) (the rule (3.101)) is applied next to the sequent in (3.106), resulting in the generation of an eigen term , z.

(M) → (h_o location z); (3.107)

The range of this z is, at the moment, NIL. Because no eigen variables exist in the TP-state, that are instances of the scope of z, namely PLACE, and are created as a result of the application of Gentzen's rules (or their adaptation to MDS). Notice that the reasoning used to assign (Range z) = NIL, is already an adaptation of Gentzen's rules, where we have taken into account the fact that variables in MDS have scopes associated with them.

At this point our objective is to construct possible ranges for the eigen term, z, using the knowledge available in (M). We shall focus on the objects and relations on the right side of (3.107), in which, z, appears. This would bring our attention to the invocation anchor, [h_o location]. There are no CC's associated with this. Thus, one is free to adopt any PLACE, as the value of (h_o location). Notice that the lack of a CC, at an anchor [$@_x$ r] has the interpretation: ((∃y)($@_x$ r y)(X r Y)(Y instance y)). In effect, we shall now apply the rule (∃ →)(the rule (3.104))

to the above predicate, to create a new instance of PLACE, say p_o, and assert (h_o location p_o) to the model space. This would cause the ASSIMILATOR to incorporate the following changes.

$$[(h_o \text{ location } p_o)(p_o \text{ locationof } (h_o\ ?))], \tag{3.108}$$

and return the following unknown residue:

$$\begin{aligned}(UR) = ([(s\ (h_o\ ?))]((s \text{ heldby NIL}) \vee \\ (s \text{ heldby:location } p_o)) \\ [(y(h_o\ ?))]((\forall y)(p_o \text{ locationof } y) \Rightarrow \\ (SAFE\ s\ y)))\end{aligned} \tag{3.109}$$

The TP interprets this UR as the condition under which (3.108) has been accepted, and incorporates this in the TP-state as follows:

$$(M)(UR) \rightarrow (h_o \text{ location } z); \tag{3.110}$$

At this point, it is possible to assign a range for z, namely, $[(z\ (p_o\ ?))]$. This would result in generating the assertion,

$$(\text{THASSERT } \sim(h_o \text{ location } z)) \tag{3.111}$$

The negation sign in (3.111) is because "(h_o location z)" appears on the right side of the sequent in (3.110), and THASSERT attempts to move "(h_o location z)" to the model space, (M), which is on the left side. As per Gentzen's rule,

$$(\rightarrow \sim):\ \frac{S_1 P \rightarrow S_2 S_3;}{S_1 \rightarrow S_2 (\sim P) S_3;} \tag{3.112}$$

the negation sign is necessary. THASSERT will seek to find a binding within the range of z, such that ~(h_o location z) for the given binding would directly contradict an existing assertion in (M). In our case, this existing assertion would be, "(h_o location p_o)". Thus z will be bound to p_o and a contradiction in the sequent would be recognized. In our example, this contradiction causes every sequent in the TP-state to be contradicted. Notice, that in the case of our example here, the TP-state has only one sequent, namely (3.110). This terminates the proof.

In the above example, we had no need to use the residue (UR) in (3.110). If the theorem to be proven had been, for example,

$$(M)(m_o \text{ location } p_o)(m_o \text{ type MISSIONARY})(c_o \text{ location } p_o)(c_o \text{ type CANNIBAL}) \rightarrow ((\text{HUMAN h})(\sim(\text{h location } p_o) \vee (\text{h type MISSIONARY})))$$ (3.113)

then (UR) would be the unknown residue at [p_o locationof]. The TP would bring this down to the sequent, on the left side. m_o, p_o, c_o, MISSIONARY and CANNIBAL will all be eigen variables. In this case, the proof will call for an expansion of (UR) and assignment of a type for h, in the model space.

In using the unknown residues from the model space in this manner, the TP attempts to complete the models in the model space in such a manner that a contradiction in the sequents may be recognized. The basis for using unknown residues in this manner is provided by part (A) of Theorem 2. In modelling the unknown residues, (UR), the TP would attempt to maintain the values of the true-partitions in the model space, associated with the (UR)'s. Completion of the models for (UR) under this hypothesis would by Theorem 2, part (A) guarantee consistency in the model space.

The model space is used in this process to actively guide the TP in seeking contradictions in the sequents. In a given TP state TP_i, each sequent in TP_i will cause a THASSERT of the form:

$$(\text{THASSERT } p_1 p_2 \ldots p_n \sim q_1 \sim q_2 \ldots \sim q_m). \qquad (3.114)$$

where the p_i's will be the literals on the left side and q_j's, the literals on the right side. If every sequent in TP_i is contradicted for the same assignment of bindings to the eigen terms, then the proof is complete. An outline of the proof procedure is presented in Appendix III. Search strategies, and problems encountered in the proof process will be discussed in a future paper.

The soundness of using a modelling facility like MDS for guiding search in a theorem proving process was independently investigated by Sanford [Sanford, D. 1977b], in the context of

a resolution based theorem proving system. The requirement of false permissive completeness of Sanford, is satisfied by the modelling schemas in MDS.

3.5.4. Basis for Means-end Analysis

The basis for means-end analysis is provided by part (B) of theorem 2. We use the following corollary for means-end analysis in model space updating processes:

Corollary 3.1:

$$[(\varphi(\Delta\alpha_f(FR([\alpha_f]P\ M_i))\ M_j) = ?) \Rightarrow (\varphi([\alpha_f + \Delta\alpha_f]\ P\ M_j) = ?)]$$

By appropriately forcing the false-residue to ?, using focus lists as a guide, a new model state M_j is obtained, in which the recognized contradictions are eliminated. It should be noted that Corollary 3.1 is true in the model space only because we have precluded occurrences of forms

$$(\varphi[\alpha_f]P = ?)$$

in any of the CC's.

Once the false-residue is set to evaluate to ?, the model space may be updated, and new values for the unknown relations may be tried from the candidates available for the relations, or by using TP, as the case may be, if such updating is found necessary.

In the case of DESIGNER, where pre-conditions are important, the above approach cannot be taken. (Because, forcing an F-residue to evaluate to ?, in effect causes the system to forget the conditions

that existed, when contradiction was recognized.) In this case, it is necessary to set new sub-goals to eliminate contradictions. In selecting these sub-goals, again focus lists may be used. The false-residue itself will appear as the initial condition for these sub-goals. These sub-goals would attempt to satisfy part (B) of Theorem 2, for $\varphi\xi = T$ or $\varphi\xi = ?$. The ramifications of this process will be discussed in a future report [Srinivasan 1977c].

IV Concluding Remarks

We have presented the logical foundations of the organization and operation of MDS. We have shown how the MDS model space is defined, how it operates, and briefly indicated how it may be used as the basis for a variety of intelligent problem solving, like assimilation, goal directed search, theorem proving, planning and recognition, and language understanding.

The physical organization of MDS may be viewed as consisting of a network of bundles. Each bundle is like a molecule. It is a basic unit of interactions and reactions. Also, it is the basis for all communication. Each bundle, corresponds in a natural way to a recognizable entity in a domain of discourse. A bundle schema represents a class of entities in the domain.

MDS itself may be viewed in terms of bundle structures. At the highest level we have the bundle, MDS itself, with component bundles, MODEL SPACE, ASSIMILATOR, DESIGNER, TP and LINGUIST. These bundles are interconnected by communication paths over which they may exchange messages in the languages of a domain of knowledge. The meta-language of MDS is used to define this domain language.

Each component bundle of MDS may itself be again described in terms of sub-bundles. The meta-language may be used to describe these component structures. The description of the MDS model space in the meta language, appears in Srinivasan [1976e].

Each bundle is associated with three kinds of entities: Data structures, Programs and Controls. The programs are mostly invoked automatically when their associated data are accessed or modified. In MDS one may not only define bundle structures, but also create instances of these structures, according to given specifications.

The bundle paradigm is a description paradigm. But, in view of the fact that MDS can instantiate bundle schemas, this description paradigm may also be used as a programming paradigm. It is not, however, a programming paradigm in the conventional sense of the phrase. We have used the phrase, Knowledge Based Programming, to refer to the kind of programming done in MDS [Srinivasan 1973b].

Central to the organization of MDS is the three valued logical system and its associated ξ-calculus. This system enables us to operate under conditions of incomplete knowledge, in a consistent way. This has also enabled us to unify under a proper pragmatic context, a whole variety of concepts and techniques known in Artificial Intelligence and Symbolic Logic, as explained in section I. We have, in fact, provided a framework for the definition of "knowledge" itself, and its associated concepts of language and description. Knowledge exists only in the context of an organizational structure. Within this structure, knowledge pertains to patterns of interactions among component units in the structure, and patterns of changes that the structures can admit. In MDS patterns of interactions are captured indirectly (via DETLISTS and DF's) by the sense-definitions, and patterns of change are captured

by transformation rules and focus-lists. At the level of domain knowledge the sense definitions portray the static laws of a domain, and the transformation rules, the dynamic laws.

Our investigations in MDS have pointed out certain ways of organizing computing machines, which would facilitate easy and efficient implementation of MDS, making use of the inherent parallel processing possibilities in the MDS architecture. These we shall discuss in a future paper.

We have perhaps raised in this paper more issues for clarification and further elucidation, than questions that we have answered, or solutions, we have proposed. The research on MDS is an on going activity. Much work remains to be yet reported, and much remains for further investigation. We have attempted to present here the logical foundations of an approach to create intelligent systems, the approach that MDS represents.

V. ACKNOWLEDGEMENTS:

This paper is based on lecture notes prepared for a seminar on MDS, during the spring of 1975. Many of the concepts, like those on filters, focus lists, commonsense reasoning, theorem proving, etc. are discussed here in greater detail. The architecture of MDS has remained the same since the spring of 1975. We have, however, had better definitions of concepts and a clearer understanding of their scope and significance.

The progress we have made on the implementation of MDS would have been impossible without the help of my students Joel Irwin, John Ng and Tau Hsu. Explaining MDS to Joel and John, clarified many of the concepts in my own mind. We would like to thank Sridharan for participating in the MDS seminar. Sri was responsible for many lively discussions, which contributed to our understanding of the significance of MDS.

An appreciation for the power, flexibility and naturalness of MDS, began to emerge only after we started our work on the BELIEVER system of Chuck Schmidt. We spent more than six months creating representations in MDS for act schemas and act interpretation schemas of BELIEVER. This laid the foundation for the design and implementation, by Sridharan and Hawrusik, of a mini-MDS system, called AIMDS. Since then, MDS concepts have profoundly influenced the scope and direction in the BELIEVER project.

I learnt about the details of Gentzen's system of logic from lectures given by Gerald Darlington and Ann Yasuhara, in the MDS seminar. This enabled me to adapt this system of logic for Theorem Proving in MDS. We gratefully acknowledge the help of Darlington and Yasuhara.

The discussion I had with Robert Balzer made me realize the potential for the application of MDS to develop "Automatic Programming systems." The concept of "Programming over a knowledge base," grew after these discussions.

The work on MDS started late in 1971, as a part of my efforts to implement a design-aid system for computing systems design, using as a basis the Computer Description Language, called CDLI [Srinivasan 1967a, b]. This work enabled me to perceive the severe problems and difficulties involved in creating an intelligent design-aid. The tyranny of programming was unbearable, and the brittleness of the resulting system would have made it obsolete, by the time it was ready. The meta system architecture of MDS came into being as a direct result of my attempts to cope with these problems. Gavin Clowe implemented the first version of a template establishment system, called TEMPEST. Since then MDS has steadily moved in the direction of being a formalism for the description and automatic utilization of knowledge.

I am thankful to Saul Amarel, who as the leader of the research group at RCA Laboratories, was responsible for getting

support for the work on CDL1, during the late 60's. The support given by RCA and the grants from the Air Force Cambridge Research Laboratories, Bedford, Mass., together made possible the work on CDL1, which later led to the development of MDS.

Since June 1971, through September, 1975, the work on MDS was supported by the National Institute of Health of the U.S. Government. During the period, March 1973 through August 1977, we had overlapping support also from the Advanced Research Projects Agency, of the Department of Defence. The support of both these institutions had been invaluable.

The writing of this paper would not have been possible but for the sabbatical leave granted to the author by Rutgers University for the year 1977. I am thankful to Saul Amarel, and to my colleagues in the department for recommending this leave and to Rutgers for approving it.

VI: References:

1. Beth, E.W. [1959] "The Foundations of Mathematics", North Holland, Amsterdam.

2. Irwin, J. and Srinivasan, C.V. [1975]: "The Description of CASNET in MDS," RUCBM-TR-49, Department of Computer Science, Rutgers University, New Brunswick, N.J. 08903.

3. Hewitt, C. [1972]: "Description and Theoretical Analysis (using schemata) ... robot", Ph.D. Dissertation, Dept. of Mathematics, M.J. Cambridge, Mass.

4. Kanger, Stig. [1963]: "A simplified proof for elementary logic", In Computer Programming and Formal Systems, Braffort and Hirschberg (Eds), North Holland, Amsterdam.

5. Le Faivre, R. [1976]: "Procedural representation in a Fuzzy problem solving system", Proc. of National Computer Conference, N.Y. 1976, pp. 1069-1074.

6. Minsky, M.A. [1975]: "A framework for representing knowledge", in Winston, O. (Ed.). The Psychology of Computer Vision", McGraw Hill, N.Y. 1975.

7. Newell, A, et. al. [1959]: "Report on a General Problem Solving Program for a Computer," in Proc. Int. Conf. on Inf. Processing, UNESCO, Paris, France, pp. 256-264, also reprinted in Computer & Automation, July 1959.

8. Sanford, D.M. [1977a]: "Hereditary-lock resolution: A resolution refinement strategy combining Strong Model Strategy with Lock Resolution," SOSAP-TR-30, Computer Science Dept., Rutgers University.

9. Sanford, D.M. [1977b]: "Formal specifications of Models for Semantic Theorem Proving strategies," SOSAP-TR-32, Dept. of Computer Science, Rutgers University.

10. Schmidt C.F., et. al. [1976]: "Recognizing Plans and Summarizing Actions," Proc. AISB, Edinburg, Scotland, pp. 291-306.

11. Sridharan, N.S. and Schmidt, C.F. [1977]: "Knowledge-Detected Inference in BELIEVER," CBM-TR-75, Computer Science Dept., Rutgers Univ. Jan. 1977.

12. Sridharan, N.S. [1976]: "The Frame and Focus Problems. Discussion in relation to BELIEVER system", Proc. AISB 1976, Edinburg, Scotland, pp. 322-333.

13. Sridharan, N.S. and Hawrusik, F. [1977]: "Representation of Actions and Transformations in AIMDS," CBM-TR-76, Computer Science Dept., Rutgers Univ.

14. Srinivasan, C.V. [1973a and 1976f]: "Architecture of Coherent Information System: A General Problem-Solving System," Proc. IJCAI3, 1973; Republished in IEEE Trans. on Computers, Vol. C-25, 4, pp. 390-402, April 1976.

15. Srinivasan, C.V. [1973b]: "Programming over a knowledge Base: The basis for automatic programming," SOSAP-TM-4, Dec. 1973, Dept. of Computer Sc., Rutgers Univ.

16. Srinivasan, C.V. [1974]: "Description in MDS of a Coherent Information System for a Banking domain," SOSAP-TM-4A, June 1974. Dept. of Computer Sc., Rutgers University.

17. Srinivasan, C.V. [1976c]: "Theorem Proving in the Meta Description System," SOSAP-TR-20, Dept. of Computer Sc., Rutgers University, January 1976.

18. Srinivasan, C.V. [1977c,d]: "Meta Description system, Part II: DESIGNER, The goal directed problem solver," and "... Part III, the Theorem Prover," Both of these are in preparation.

19. Srinivasan, C.V.: [1967, a,b]: "Introduction to CDL1, A Computer Description Language," and "Formal Definition of CDL1," Scientific reports 1 and 2, Sept. Oct. 1967, Issued to contract AF 19(628) 4789, contract Monitor Rocco H. Urbano, Air Force Cambridge Research Laboratories, Bedford, Mass.

20. Winograd, T 1975 : "Frames and the declarative procedural controversy," in Bobrow, D.G. and Collins, A.M. (Eds.), Representation and Understanding, Academic Press, N.Y.

21. Srinivasan, C.V. [1976e]: "Formal definition of the model space of the meta description system," SOSAP-TR-20B, Dept. of Comp. Sc., Rutgers University.

APPENDIX I

A MODIFICATION TO DESCRIPTION STRUCTURE

We wish here to admit the situation where more than one person or vehicle may hold an object. We shall also make the holding relation transitive. To effect these changes, we shall change the description schema, [S8] as follows:

[S8] : (ITEMS heldby AGENTS)

[S11]: (AGENTS elements (HUMAN VEHICLE))

[S12]: ([ITEMS heldby] flag X).

"X" flag indicates that the relation is transitive.

Also, we shall associate a ne CC, with [ITEMS heldby]:

CC[ITEMS heldby]:

(y | (@ heldby y) ~(y element @)
((VEHICLE x z)(SOME w)
(w heldby x)(w heldby z) ⇒
((x heldby z) V (z heldby x)).

First of all, an item cannot hold itself. Then, if two vehicles x and z hold the same object w, then (x heldby z) or (z heldby x) should be true. Most of the arguments presented in section 2 will go through also for this new representation: Only the section on "Frame checking in the Model space," Section 2.5.7.4. would appear different. Punt.

APPENDIX II

Representation of Collection and Sets in the Model Space

1. Model Space events and the Event State.

An event in the model space is an ASSIMILATOR command that either creates a new model or updates the properties of some of the existing models. Every event is assigned an event number, in increasing order of its appearance. The event state of the model space is the event number of the current (last event). We shall use event numbers to specify the recency of values and reasons in the model space.

2. Collection and Sets

A collection (set) is represented by a list of the form:

$$(z) = (\delta \; \# \; f \; c \; u \; r \; z^1 \; z^2 \text{---} z) \qquad (1)$$

where δ, $\#$, f, c, u and r have the following interpretations:

$\delta(z)$ is the pointer to the definition of (z).

$\#(z)$ is the number of known elements in (z).

f(z) = ? or NIL is the flag of (z). (Open (z)) is true if f(z) = ? In this case (z) has room for more elements. Otherwise, (closed(z)) is true.

c(z) is the compliment of (z).

u(z) is the unknown part of (z)

r(z) are the reasons (residues) associated with (z).

We shall use (z^n) to denote a collection with exactly n elements and $(z^n\ ?)$ to denote a collection which has n elements, but has room for more.

2.1 Definition of (z)

Each collection (z) is defined by a list template. Let $\delta(z)=Z$. The frame schema (Z elements $(Y_1\ Y_2 \ldots Y_k)$), partly defines the elements of Z. The full element definition will have the form:

$$\delta(z) = (Z\ \text{elementdn}) =$$
$$(\ \langle\text{edn}\rangle_1\ \langle\text{edn}\rangle_2 \cdots \langle\text{edn}\rangle_k\ ) \qquad (2)$$

where each <edn> is an element definition of the form,

<edn> = [<lb> <ub> <scope> <cc> <atr>],

where <lb> is the lower bound on the number of instances of <scope> that (z) may have and <ub> is the upper bound on this number. The <cc> will constrain the instances of <scope> that may appear as elements of (z). The <atr> will specify rules for adding (deleting) instances of <scope> to (from) (z). The scope of (z), (Scope (z)) will be the disjunction of the scopes of its element definitions.

If Y is the <scope> of an <edn> of Z then we shall say that (Z element Y) is true. Also, we shall say that X is an instance of (Scope (z)) if it is an instance of one of its disjuncts.

2.2. Compliment of (z)

c(z) has the form

$$c(z) = (\text{"c"} \# f\ t\ r\ z^1\ z^2\ \ldots\ z) \qquad (2)$$

where "c" indicates that the list is a compliment of a (z), f(c(z)) = ? or NIL specifies whether c(z) is open or closed, t(c(z)) = z is the true part of the list, and r(c(z)) are the residues associated with the members of c(z). If x is a member of c(z) then x is not an element of the collection (z). Only instances of (Scope(z)) may appear as members of c(z).

2.3. The Unknown Part of (z)

$$u(z) = (\text{"u"} \# f\ t\ r\ z^1\ z^2 \ldots z\ ). \qquad (3)$$

If u(z) exists then its members will represent the candidates for elements of (z).

2.4. Elements of (z)

We shall use "(z)" to denote both the collection (z) and the list (z). The relation name "memberof" is used exclusively for lists, and "elementof" is used only for collections. Let CC[Z element Y] denote the CC anchored at the <edn> whose scope is Y. Then,

$$[(Z \text{ element } Y)(Z \text{ instance } (z)) \Rightarrow ((@_y \text{ elementof } (z)) \Longleftrightarrow CC[Z \text{ element } Y]((z)\ @_Y)] \qquad (4)$$

The representation of (z) is defined by the following:

(a) $(@_y$ elementof $(z)) \Leftrightarrow$
$((@_y$ memberof $(z))$ V
$(\sim(@_y$ memberof $c(z))$
$((\text{Closed } c(z))$ V $\sim(@_y$ memberof $u(z))))$

(b) $\sim(@_y$ elementof $(z)) \Leftrightarrow$
$((@_y$ memberof $c(z))$ V
$\sim(@_y$ instanceof $(\text{Scope } (z)))$ V
$(\sim(@_y$ memberof $(z))$
$((\text{Closed } (z))$ V $\sim(@_y$ memberof $u(z))))$

(c) $((@_y$ elementof $(z)) = ?) \Leftrightarrow$
$((@_y$ memberof $u(z))$ V
$\sim(@_y$ memberof $(z))$ V
$\sim(@_y$ memberof $c(z)))$
$(@_y$ instanceof $(\text{Scope } (z))$
$(\text{Open } (z))(\text{Open } c(z)))$ (5)

Each collection thus defines a partition of all the items in its scope. Let $\Pi(z)$ denote the partition defined by (z):

$$\Pi(z) = [T(z)\ U(z)\ F(z)] \tag{6}$$

where

(a) $T(z) = (x \mid (x \text{ elementof } (z)))$

(b) $U(z) = (x \mid (x \text{ elementof } (z)) = ?)$

(c) $F(z) = (x \mid \sim(x \text{ elementof } (z)))$ (7)

Notice that an unknown element of a collection may exist only if both (z) and c(z) are open. If x is an unknown element of (z) then x may potentially belong either to (z) or to c(z).

In general, the unknown truth value of a relation, in the MDS model space, is a property that is incidental to the model space. It is not a property that is intrinsic to the domain, that is being modelled. Thus, one may have relations that are universally true (false), i.e. true (false) in all possible model spaces of a domain. But, there is no notion of an universally unknown truth value*.

The element definitions given in (3.6) enable economical representations of collections in the model space. This is discussed below:

2.5: Special Cases of (z)

(i) Instances of classes

Let I(Y) denote the collection of all instances of a class Y. The form of I(Y) is,

$$I(Y) = (Y\ \#\ f\ e\ r\ y^1\ y^2 \ldots y) \qquad (8)$$

where e(I(Y)) is the event number of the last event, that created an instance of Y. In this case the complement of I(Y) is not stored. By definition,

$$(y \text{ instanceof } Y) \Longleftrightarrow (y \text{ memberof } I(Y)),$$
$$\sim(y \text{ instanceof } Y) \Longleftrightarrow \sim(y \text{ memberof } I(Y)) \qquad (9)$$

---

* The Theorem Prover in MDS operates in two valued logic. But it uses the model space, which is in 3-valued logic.

(ii) NULL Collections

(NULL (z)) is true if

(z) = (δ 0 NIL (y))

(y) = (c e NIL (z) r). (10)

Here (z) has no members: Its cardinality is 0 and it is closed. The "e" in the compliment of (z) indicates that all instances of (Scope (z)) as of the event, e, are members of (y). Every time (z) is accessed this event number, e, will be compared with the event numbers associated with the classes in the scope of (z). If any of these classes has an event number that is greater than e, then the corresponding elements of (z) will be computed, using the cc's and ATR's associated with (z). If all the event numbers of the classes are less than or equal to e, then the existing specifications of (z) will be used.

We shall use

(z) = δ(z)[NIL NIL T] (11)

to denote the NULL collection.

(iii) UNIVERSAL Collections

(ALL (z)) is true if

(z) = (δ e NIL (y) - r)

(y) = (c 0 NIL (z)) (12)

Here again c, is given the same interpretation with respect to the list (z), as in (ii) above. We shall use

$$(z) = \delta(z)[T\ NIL\ NIL] \tag{13}$$

to denote the universal collection.

(iv) Open Collections

As a collection (z) is open only if as lists, both (z) and c(z) are open. Open collections may have one of the following forms:

a) $\delta(z)[T(z)\ ?\ F(z)]$,
b) $\delta(z)[T(z)\ ?\ ?\ ]$,
c) $\delta(z)[\ ?\ ?\ F(z)]$,
d) $\delta(z)[\ ?\ U(z)\ F(z)]$,
e) $\delta(z)[\ ?\ U(z)\ ?\ ]$,
f) $\delta(z)[T(z)\ U(z)\ ?\ ]$, or
g) $\delta(z)[\ ?\ ?\ ?\ ]$. (14)

As we shall see below, it will never be necessary to enumerate in a collection all the items in its scope. Compared to the number of instances in the scope of a collection, the number of members in the representation of the collection will always be small.

(v) Values of $(@_x\ r)$

We shall use collections to represent the values of relations

$(@_x\ r)$. Every $(@_x\ r)$ thus defines a partition of the items in its scope. If (X r Y) is true and Y is a node template then the value of $(@_x\ r)$ will be one of the following:

$$(@_x\ r) = (z) = (\delta\ 1\ \text{NIL}\ (y) - r\ y)$$
$$(y) = (c\ ?\ ?\quad (z)) \tag{15}$$

if $(@_x\ r\ y)$ is true Or,

$$(@_x\ r) = (z) = (\delta\ 0\ ?\ (y)\ (u))$$
$$(y) = (c\ e\ ?\ (z))$$
$$(u) = (u\ n\ f\ (z)\ r\ z^1 \ldots z^n) \tag{16}$$

if candidates $(z^1, z^2, \ldots, z^n)$, are known for $(@_x\ r)$, but $(@_x\ r)$ itself is unknown. Or

$$(@_x\ r) = (z) = (\delta\ 0\ ?\ (y)\ u))$$
$$(y) = (c\ n\ ?\ (z)\ r\ y^1 \ldots y^n)$$
$$(u) = (u\ e\ ?\ (z)) \tag{17}$$

if $(@_x\ r)$ is unknown but some y, for which $\neg(@_x\ r\ y)$ is true, are known

Or

$$(@_x\ r) = (z) = (\delta\ 0\ ?) \tag{18}$$

if nothing is known about $(@_x\ r)$. Similar representations will exist also for the case where Y is a list template in (X r Y). The partitions implied by these representations are made use of to obtain complete solutions of Consistency conditions. The CC evaluation algorithm is discussed in section 3.4.

Appendix III

Adaptation of Gentzen's system of Logic to Theorem Proving in MDS.

The generalization and instantiation rules have been already explained in the text (see (3.101) through (3.104)). We also have the following propositional rules:

Splitting Rules:

$(V \rightarrow)$: $\dfrac{S_1(P)S_2 \rightarrow S_3;\ S_1(Q)S_2 \rightarrow S_3;}{S_1(PVQ)S_2 \rightarrow S_3;}$

$(\rightarrow \wedge)$: $\dfrac{S_1 \rightarrow S_2(P)S_3;\ S_1 \rightarrow S_2(Q)S_3;}{S_1 \rightarrow S_2(P \wedge Q)S_3;}$

Absorption Rules:

$(\rightarrow V)$: $\dfrac{S_1 \rightarrow S_2(P)(Q)S_3;}{S_1 \rightarrow S_2(PVQ)S_3;}$

$(\wedge \rightarrow)$: $\dfrac{S_1(P)(Q)S_2 \rightarrow S_3;}{S_1(P \wedge Q)S_2 \rightarrow S_2;}$

Negation Rules:

$(\sim \rightarrow)$: $\dfrac{S_1S_2 \rightarrow S_3(P);}{S_1(\sim P)S_2 \rightarrow S_3;}$

$(\rightarrow \sim)$: $\dfrac{(P)S_1 \rightarrow S_2S_3;}{S_1\ S_2(\sim P)S_3;}$

For a given TP-state all applicable propositional rules should be applied first. Thereafter, the <u>needed</u> generalization and instantiation rules are applied to a TP-state, TP_i, to generate the next TP-state, TP_{i+1}. In the state TP_{i+1} for each sequent, say s_j^{i+1}, $1 \leqslant j \leqslant k$, the collection of assertions

$$\alpha(s_j^{i+1}) = (p_{j1} \cdots p_{jn} \sim q_{j1} \sim q_{j2} \cdots \sim q_{jm})$$

is constructed, and all the THASSERT's,

$$(\text{THASSERT } \alpha(s_j^{i+1})), \quad j = 1,2,\ldots,k,$$

are asserted to the Model Space in parallel. This might result in the recognition of contradictions in each sequent of TP_{i+1}. In such a case the TP process would terminate. Otherwise, the unknown residues generated by the ASSIMILATOR in response to the THASSERT's will all be brought to their respective sequents and the TP-state, $\dot{TP}_{i+1}$, will be updated. In doing this the following rules will be followed.

The UR associated with a p_i will be brought down to the left side of its associated sequent, and the UR associated with a $\sim q_j$ will be brought to the right side. It should be noted that application of an instantiation rule also may result in the receipt of an UR, returned by the ASSMILATOR as the conditions for the acceptance of the instantiation. The UR's associated with the instantiation rule $(\exists \rightarrow)$ will go to the left side of a sequent, and those associated with $(\rightarrow \forall)$ will go to the right side.

Each THASSERT will cause the ASSIMILATOR to call in its own checking and updating processes. This may result in narrowing down the ranges for the eigen terms in the TP-state.

One may assume that the variables in each UR entered in a sequent would be distinct and new to the TP-state. Also, in each UR the following expansions would have been incorporated, thereby removing the binding expressions appearing in the UR:

$$[(z\ (a_1\ a_2 \ldots a_n\ ?))]\ ((\exists z)P_1P_2\ldots P_k) \text{ and}$$
$$((\exists z)[(z(a_1a_2\ldots a_n\ ?))]P_1\ P_2\ldots P_k)$$

are transformed to

$$\bigvee_{i=1}^{n}[(z\ a_i)]P_1P_2\ldots P_k)\ \ V\ \ ((\exists z)P_1P_2\ldots P_k),$$

and all subexpressions of the form,

$$((\forall z)(\bigvee_{i=1}^{k}(z(a_1^i\ a_2^i\ \ldots a_n^i\ ?))]P_i)$$

are transformed to

$$(\bigvee_{i=1}^{k}(\bigwedge_{j=1}^{n}([(z\ a_j^i)]P_i)))((\forall z)(P_1\ V\ P_2\ V\ldots V\ P_k)).$$

In both the above expansions the right most quantified expression will occur only if the binding ranges for z include a ?.

In attempting THASSERT if the range for an eigen term, z, is NIL, then

a) either further applications of appropriate instantiation rules would be sought, or

b) the model completion procedure for the relevant z's would be initiated.

An example of the model completion task was encountered in the discussion in section 3.5.2. There are also cases, in which model completion procedure should be initiated in order to find new possible bindings for an eigen term, z, even though it has already a non-NIL range. The model completion as a strategy for using domain knowledge will be discussed in a future report.

.....