THE ARCHITECTURE OF COHERENT INFORMATION SYSTEM: A GENERAL PROBLEM SOLVING SYSTEM*

C. V, Srinivasan
Department of Computer Science
Rutgers University-
New Brunswick, New Jersey 08903

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 a large and expanding data base. 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.

1. Introduction

Our objective is to create a meta-system which can be used to generate intelligent information systems in different domains of discourse. The' meta system is called the META DESCRIPTION SYSTEM CMOS). It has facilities *to* accept definitions of description schemas and descriptions themselves, of"KNOWLEDGE -- about facts, objects, processes, and problem solving -- in a domain. A domain might be a disease system, a piece of mathematics, or computing systems themselves. The description schemas and descriptions of knowledge in a domain specialize the MDS to act as an intelligent information system for the domain. For a domain M, the information system associated with it is called the COHERENT INFORMATION SYSTEM of M.

In our research we have two principal concerns: (i) How may one describe knowledge in a domain to a computer; what kinds of knowledge should a system have to exhibit intelligent behaviour; what operational facilities are needed to accept and use such knowledge? (ii) How *may* the computer be made *to* use given knowledge automatically to solve problems in the domain and answer questions?

The MDS accepts and uses three kinds of knowledge: a) Structural knowledge pertaining to the form and syntax of descriptions. Descriptions may, of course, be strings of words in some language. The MDS will translate such descriptions to structures within a relational system. The relational system itself may consist of constants, variables, predicate symbols, function symbols, logical operators and quantifiers. The structural knowledge specifies the structure of the relational system used in a domain. b) Sense knowledge: Logical assertions pertaining to the sense in which structures are interpreted, and constraints on admissable structures beyond those specified in the syntax. And, c) Transformational knowledge: This pertains to the knowledge necessary to transform given descriptions of specific objects to new ones, according to specified criteria.

Corresponding to these three levels of knowledge there is a hierarchy of problem solvers, (CHECKER, INSTANTIATOR), THEOREM PROVER (TP) and DESIGNER, in order of increasing complexity. The (CHECKER, INSTANTIATOR) system acts as a sophisticated data management system that establishes, maintains and updates the data base of models of specific objects in a domain in a manner consistent with the structural and sense knowledge. CHECKER can answer questions pertaining to any of the specific models for which the information is either directly stored in the data base, or is directly derivable by evaluating a given logical assertion in a given context. The THEOREM PROVER adds power to the CHECKER in three ways: In certain cases it helps reduce the search effort of CHECKER by giving it advice based on deduced consequences of sense knowledge; where feasible it can warn the CHECKER of impossible situations in the generation and updating of models; it can also determine general truth values of assertions based on the structure and sense knowledge. The DESIGNER adds further power to the system by enabling the system to plan courses of actions using given action primitives (Transformation Rules} in a manner consistent *with* the facts *of* a problem. This hierarchy imposes a very useful classification of system facilities, and gives the system a considerable flexibility.

The descriptive language of a domain is itself specified in terms of the model definitions in the domain. Language analysis is thus looked at as a model building process. Most importantly, the model definitions in a domain may include definitions of Problem Solving States (PSS), relevent to the domain. The PSS may provide facilities to summarize the problem solving experience of the system. This summary may be used to intelligently guide the problem solver.

This work on MDS and CI-systems may be thought of essentially as a further extension of the trend started by REF-ARF[I,2], QA4 [3], POPS [41, STRIPS [5,61, and PLANNER [7], Its problem solving activity uses "means-end" analysis, a concept originally introduced in GPS [8], and function invocation schemes based on goals, introduced by PLANNER. Ct-Systems have both the flexibility of PLANNER-like systems, and model based reasoning abilities of a GPS like system. The entire system depends on the way descriptive data structures are organized in a given domain. However, the availability of data structure and model definition facilities, and a separate data management system makes it possible to completely isolate the data structure and data base details from the problem solving programs. This makes it possible to conceive of the meta system, the MDS, to create CI-Systems for different domains. It seems reasonable that, if the classes of possible models of objects in a domain could he described to a computer then, in principle, the computer should be able to make use of the descriptions for problem solving and language understanding in the domain. In CI-Systems we show how (a) classes of models can be defined and (b) how the definitions could be used for language analysis and problem solving in the domain.

? This work was supported by a research grant from NIH, grant number RR-643,

The principal contributions of the proposed architecture are:

i) A facility to use large data bases;

ii) A stratification of knowledge in a domain and the facility to use a highly flexible descriptive mechanism to describe objects and problems in a domain; the possibility of describing knowledge in a domain in a systematic way to a computer;

iii) The definition of the descriptive language itself in terms of the models the system can build in a domain; and

iv) The possibility of specializing the MDS to operate efficiently as a problem solving system in a domain of discourse.

The MDS is now being implemented in LISP 1.6. Some parts of it (see Section 3) are now ready. This paper is, therefore, a report on work currently in progress. It introduces the principal architectural concepts of MDS and CI-Systems in the context of an example, the Missionaries and Cannibals* (M&C) problem [9]. The structure of CHECKER and DESIGNER is explained. The operation of the THEOREM PROVEN is discussed in [10]- In a subsequent paper the language processor will be discussed.

2. An Overview of the System Architecture

2.1. Templates and Their Instantiations

2.1.1. The Templates

The concept of TEMPLATES, the devices used to specify structural knowledge is central to the entire system architecture. Templates classify objects in a domain into objects of different kinds and types. Each template specifies a certain description structure. Thus, in the M&C problem (see Table 1) PLACE, PEOPLE, VEHICLE, etc. are different kinds of objects. The template for PLACE, for example introduces two relation symbols: occupants and position of. The pair of relation symbols (occupants, occupants of) for example, are inverses of each other in the sense that in instances of PLACE and PEOPLE the relations (PLACE occupants PEOPLE) and (PEOPLE occupants of PLACE) will always appear together in the data base of models. PEOPLE is just a list of PERSONS. An instance of type classification occurs in the PERSON template. A PERSON can be a MISSIONARY or CANNIBAL. In MDS type classification always reflects distinctions in the way objects are used. The templates thus specify the structure of the relational system for a domain: the relation symbols to be used in the description of various kinds of objects in the domain, and the kinds of objects that a relation symbol may relate.

Given such templates, one may use the INSTANTIATOR to create descriptions, which are instances of the templates. Such instances might be specified to the system in some external language, which is translated to the internal representation in the relational system. Or, the system itself might generate an instance of a template when called upon to do so, In either case, to complete the instantiation of 2 template, all the relation symbols defined for the template should be assigned values. These values will

TABLE I: TEMPLATES FOR THE M&C PROBLEM

```
1. PLACE:  (occupants PEOPLE occupants of), CC1
           (position of VEHIL position), CC2
2. PEOPLE:  (elements PERSON elements of)
3. VEHIL:  (elements VEHICLE elements of)
4. PERSON:  (type PTYP type of)
            (occupant of PLACELI occupant), CC3
5. PTYP:  MISSIONARY, CANNIBAL
6. PLACEL1:  (elements (PLACE, VEHICLE) elements of)
7. PLACEL:  (elements PLACE elements of)
8. VEHICLE:  (pilots PEOPLE pilots of)
             (position PLACE position of)
             (cango to PLACEL destination of)
             (capacity INTEGER capacity of)
             (occupants PEOPLE occupants of), CC4
[CC1]  (*! occupants ((PEOPLE X)(*! occupants X)
                     (((NUMBEROF MISSIONARY X)2
                       (NUMBEROF CANNIBAL X))v
                      ((NIJMBEROF MISSIONARY X) is 01)))
[CC2]  (*! position of ((VEHICLE X)(M position of X)
                        (X cango *]))))
[CC?>] (*! occupants of.#.is 1)
[CC41  (*! occupants.".-■.capacity of *!)
```

be specific instances of objects within the data base.

Thus for the MfC problem one may create instances of PLACli's called RBANKI and RBANK2, a VEHICLE called BOAT, and as many MISSIONARIES and CANNIBALS as necessary. Each PERSON will be the occupant of some PLACE and the VEHICLE itself will be at one of the PLACES. We have not, however, introduced any of the conditions of the problem. Not all instantiations of the templates of the M&C problem would represent legal situations. The necessary additional constraints are introduced by the *sense* knowledge. Every relation symbol in a template may have a Consistency Condition (CC) associated with it. CC1 in Table I is associated with the symbol "occupants". It says that the CANNIBALS at a PLACE cannot outnumber the missionaries. The symbol "*'" in CCI refers to the current instance of PLACE at which the CC might be evaluated. It is called the anchor; (PEOPLE X) stands for "(VX)(X is PEOPLE)". All CC's have the form: "(*! r P(X)}" where *! is the anchor, r is a relation symbol occurring in the template associated with *!, and P(X) is some logical predicate. The predicate P(X) is said to be anchored at the (template, relation symbol) pair. Thus, the predicate in [CO] is anchored at (PLACE, occupants).

In [CCI] notice that "(*! occupants X)" is itself a term in its predicate. This has the following significance: For a PLACE like, say RBANKI, if the system is told to set (RBANKI occupants y) for *soma y,* it would first construct the combined list of existing occupants of RBANKI and *y,* and then verify the predicate. CC's of this kind are called declarative CC's, as opposed to the other kind, called imperative CC's, like, say (for a hypothetical template PERSQN1)

```
[CS1]  (*I sibling ((PERSON) X)(NOT ( X is *!))
                   (X child of.father of *!)))
```

[CS1] may be used to find the siblings of a PERSON1 in terms of the child of and father of relation symbols. The CHECKER is used to evaluate CC's. We shall discuss the evaluator in Section 2.2,

The significant points to be noted about CC's are the following:

(i) the knowledge represented by the CC's is of a different kind from the structural knowledge, specified by the templates.

#There are three missionaries and three cannibals on one bank of a river. They want to go to the other bank. There is only one boat available. It can carry only two people at a time. The cannibals at a shore should not outnumber the missionaries at the same shore. Find a way of transporting them.

(ii) Each CC is specifically associated with a particular relation symbol. A relation symbol, say "likes", might be quite different in the context (HUMAN likes SOMETHING), from (CATTLE likes SOMETHING). A CC is invoked and interpreted only within the particular local context of its anchor, within the overall structure of descriptions.

(iii) The logic of the CC's is highly dependent on the structures specified by templates. Also, for a given system of templates there may be more than one way of choosing and anchoring the CC's. Further, for a given domain, there will undoubtedly be several ways of defining the templates and its associated CC's. These different definitions will correspond to different ways of representing the knowledge in the domain. The MDS provides facilities to experiment with different choices. At present we have no formal guidelines to make these choices intelligently. The particular choices made in a domain will have an effect on system efficiency.

2.1.2 Instantiation of Templates

We shall call an instance of a template as the model of the object instantiated. Thus, the model of RBANK1 will be an instance of PLACE. Every triplet (x r y) (where r is a relation symbol) appearing in the model x should be dimensionally consistent: That is, for some templates M and T, where x is an instance of M and y is an instance of T, either (M r T) occurs in M, or (T $\overleftarrow{r}$ M) occurs in T, where $\overleftarrow{r}$ is the inverse of r. There are a few relation symbols which are system wide, like template of, name of, elements of, arguments of etc., which can appear with all instances in the data base, and need not be defined in the templates.

The model of RBANK1 will be a vector of five pointers, say $(P_{to},P_n,P_{eo},P_o,P_{po})$ corresponding to the relations template of, name, elements of, occupants and position of, respectively. P_{to} will point to a pair (P^1_{to},P^2_{to}), where P^1_{to} points to the PLACE template, and P^2_{to} to possibly local conditions (LC's) associated with RBANK1. P_n will, of course, point to "RBANK1". Let r be any one of the remaining relations: P_r will point to a quintuple of the form $(\#,P^1_r,P^2_r,P^3_r,P^4_r)$, called the descriptor unit of P_r (or $\overleftarrow{r}$). The elements of the descriptor unit are the following:

Descriptor Unit

- P^4_r: *Pointer to y such that (RBANK1 r y) is true,* or pointer to list (y) such that (RBANK1 r z) is true for every $z \in y$. We shall write this as (RBANK1 r (y)).
- P^3_r: Pointer to list (y) such that for every $z \in y$, (NOT(RBANK1 r z)) is true.
- P^2_r: To local conditions on values of (RBANK1 r).
- P^1_r: To TR's (Transformation Rules) local to (RBANK1 r), called LTR's.
- #: The number of elements in the list, set or triplet pointed to by P^4_r.

Every P^i_r will have an inverse, say $\overleftarrow{P}^i_r$, which will point back to RBANK1; $\overleftarrow{P}^4_r$ is the same as $P^4_{\overleftarrow{r}}$. The inverse of (P^1_{to},P^2_{to}) will be $(\overleftarrow{P}^1_{to},\overleftarrow{P}^2_{to})$, where $\overleftarrow{P}^1_{to}$ is the same as P_i (i for instance); P_i will point to RBANK1 from PLACE template.

A pointer in a model can have one of four values: NS(Not Stored), NEI (Not Enough Information), NIL, or an address (or value). Initially all pointers in a model are set to NEI. A list, set or tuple will have NEI as an element if it is incomplete. Templates thus specify the data structures of models in a domain. They provide the basic framework for the organization of domain dependent knowledge. They also play a major role in the specification and use of problem solving programs in a domain, as we shall see in Sections 2.2 and 2.3.

There are about fifteen different kinds of templates in the MDS. Variations in the structure of descriptions may be specified by defining, what are called variable templates. Exceptions to the CC's may be specified by associating local conditions *(LC's)* with specific instances of templates. An *LC* may be a conjunctive LC (CLC) or a disjunctive LC (DLC). A model should satisfy $((CC \wedge CLC) \vee DLC)$ at each one of its relation symbols. Similarly, transformation rules (TR's) for changing a model may be local to a model (LTR) or may be associated with templates themselves.

In addition to the CC's associated with pairs (M, r), where M is a template and r is a relation symbol, r may also have Properties (PR) defined for it, which apply to all occurrences r within the relational system. A typical such property is the transitivity property. For a model m, the PR's are used to identify objects y, such that (m r y) is true, but is not stored in the data base.

All problem solving programs communicate with the data base via the INSTANTIATOR and CHECKER. Templates also provide a way of classifying and storing the CC's, LC's, TR's and LTR's. Every CC (LC) is anchored at (m, r) where m is a template (or model) and r is a relation symbol (relation) defined on m. The DON-list of a CC (LC, TR, or LTR) is the list of (m_i,r_i) on which it depends. The DET-list of a pair (m,r) is the list (m_i,r_i) of pairs which depend on (m,r). So also, the DET-list of a TR (LTR) is the list of (m_i,r_i) which are affected by the TR (LTR). The DON and DET lists are stored with each CC, LC, TR and LTR. Also, every pair (m, r) will have a pointer to its associated CC, LC, TR and LTR.

2.2 Evaluation of Consistency Conditions

2.2.1. The Logic of CHECKER

The evaluation of CC's, LC's and PR's will involve searching of the models in data base. The conditions themselves specify the search paths. The anchor "*!" may be used to optimize this search. One can write small efficient programs (say, about 3K PDP-10 words of compiled LISP code) to evaluate these conditions. (Alternatively one may compile each CC and PR individually.) The CHECKER is the interpreter for CC's, LC's and PR's. It uses the G function (G for Get) of the INSTANTIATOR to retrieve objects from data base. G does the following:

(Q1) G(X r ?) = $\{y \mid (x\ r\ y)\}$ (This may include NEI), or NIL or NEI.
(Q2) G(X r y) = YES, NIL or NEI
(Q3) G(X ? y) = p, NIL

where p is a relation path $(r_1,r_2,\ldots,r_k)$ joining X and Y (the shortest one). G just looks up the data base using the templates. If the answer is NS, NEI, or if NEI is included in the answer, then G will invoke the CHECKER to evaluate the associated CC's, *LC's and PR's. This evaluation may cause the NEI to* be removed and possibly add new elements to (y).

The CHECKER operates on a three valued logic system having truth values T (TRUE), F (FALSE) and ? (NEI). The logic of CHECKER is shown in TABLE II. For a given predicate P, besides returning its logical value (one of T, ?, F), the CHECKER also returns seven other quantities, all of which will be *subexpressions of P. These are explained below.*

TABLE II: LOGIC OF CHECKER

Literal x ϕ_x denotes the value of x.

ϕ_x	$TR_\phi(x)$	$FR_\phi(x)$	$R_\phi(x)$	$TP_\phi(x)$	$FP_\phi(x)$	$NTP_\phi(x)$	$NFP_\phi(x)$
T	x	T	T	x	T	T	x
?	?	?	x	?	?	x	x
F	F	x	F	F	x	x	F

$\wedge$	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

$\vee$	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

ϕ_x	$\sim\phi_x$
T	F
?	?
F	T

Propositions: P, Q. Let X denote one of TR, FR, R, TP, FP, NTP, NFP. Then

$[X_\phi(P \wedge Q) = \sim X_\phi(\sim P \vee \sim Q)]$ is true.

Also "$\wedge$" and "$\vee$" are symmetric:

$X_\phi(P \wedge Q) = X_\phi(Q \wedge P)$; $X_\phi(P \vee Q) = X_\phi(Q \vee P)$.

The various functions are defined below for $(P \wedge Q)$.

$(P \wedge Q)$	(ϕ_P, ϕ_Q)					
	(T, T)	(T, ?)	(T, F)	(? ?)	(?,F)	(F,F)
TR	$[TR_\phi(P) \wedge TR_\phi(Q)]$	?	F	?	?	F
FR	T	?	$FR_\phi(Q)$	?	$FR_\phi(Q)$	$[FR_\phi(P) \wedge FR_\phi(Q)]$
R	T	$R_\phi(Q)$	F	$[R_\phi(P) \wedge R_\phi(Q)]$	F	F
TP	$[TP_\phi(P) \wedge TP_\phi(Q)]$	$TP_\phi(P)$	$TP_\phi(P)$	?	F	F
FP	T	?	$FP_\phi(Q)$	?	$FP_\phi(Q)$	$[FP_\phi(P) \wedge FP_\phi(Q)]$
NTP	T	$NTP_\phi(Q)$	$NTP_\phi(Q)$	$[NTP_\phi(P) \wedge NTP_\phi(Q)]$	$[NTP_\phi(P) \wedge NTP_\phi(Q)]$	$[NTP_\phi(P) \wedge NTP_\phi(Q)]$
NFP	$[NFP_\phi(P) \wedge NFP_\phi(Q)]$	$[NFP_\phi(P) \wedge NFP_\phi(Q)]$	$NFP_\phi(P)$	$[NFP_\phi(P) \wedge NFP_\phi(Q)]$	$NFP_\phi(P)$	F

In all the following functions assume that ϕ is a particular valuation of the variables in P. $\phi(P)$ = T, F or ?.

$TR_\phi(P)$:	True Residue of P. The subexpression of P that caused *$\phi(P)$ = T.*
$FR_\phi(P)$:	False Residue of P. The subexpression of P that caused $\phi(P)$ = F.
$R_\phi(P)$:	Residue of P. The subexpression of P that caused $\phi(P)$ = ?.
$TP_\phi(P)$:	True Part of P. The subexpression of P that evaluated to T. $\phi(P)$ itself may be T, F, or ?.
$FP_\phi(P)$:	False Part of P. The subexpression of P that evaluated to F. $\phi(P)$ itself may be T, F, or ?.
$NTP_\phi(P)$:	Not True Part of P. The subexpression of P that evaluated to F or ?. $\phi(P)$ itself may be T, F, or ?.
$NFP_\phi(P)$:	Not False Part of P. The subexpression of P that evaluated to T or ?.

The various residues and parts of P are used in the planning phase of DESIGNER to summarize past experiences of the system. In the THEOREM PROVER the residues are used to guide the problem solving search. The properties of these functions that make them useful are given below. The logic of these functions is shown in Table II.

Let ϕ and ψ denote valuations of the variables in P. Assume in what follows, that a certain task k was done either by the THEOREM PROVER or DESIGNER, *and the outcome of the task k depended on P. Also, at the time k was attempted,* ϕ was the valuation of the variables in P. At a later time, a new valuation ψ of the variables P was obtained, and the system had to decide whether to attempt k for the new valuation.

1. $(\forall P)(\forall\phi)(\forall\psi)(\phi(TR_\psi(P)) \Rightarrow \phi(P))$.

If $\phi(TR_\psi(P))$ is true then try k again.

2. $(\forall P)(\forall\phi)(\forall\psi)(\sim\phi(FR_\psi(P)) \Rightarrow \sim\phi(P))$.

Do not try k if $\phi(FR_\psi(P)) = F$.

3. $(\forall P)(\forall\psi)(\exists\phi)((\phi(NTP_\psi(P)))(\phi(TP_\psi(P)) \Rightarrow \phi(P))$.

A goal k could not be reached for valuation ψ because $\psi(P) = F$. Then try and find a ϕ for which $\phi(NTP_\psi(P))(\phi(TP_\psi(P))$ *is true. This is used to break* up a goal into subgoals, in means-end analysis.

4. $(\forall P)(\forall\psi)(\exists\phi)((\phi(R_\psi(P)))(\phi(TP_\psi(P))) \quad \phi(P))$

Here goal k could not be reached because $\psi(P) = ?$, *and the unknown parts of P is given by $R_\psi(P)$. Then* find a ϕ (build new objects) for which $\phi(R_\psi(P))$ and $(\phi(TP_\psi(P))$ are true. This is used in means-end analysis, and in the THEOREM PROVER.

2.2.2. What CHECKER and INSTANTIATOR can do; What more is needed

The CHECKER and INSTANTIATOR together act as a fairly sophisticated data base management system.

The CHECKER makes sure that data entered into the data base is consistent, and also keeps track of what additional data is needed to complete the descriptions of objects with respect to the templates. The templates for a domain describe the structure of the data base for the domain. The CHECKER uses this structure to guide the INSTANTIATOR to create and retrieve items in the data base selectively.

The limitations of the CHECKER arises in the automatic guidance it can provide in the updating process. The CHECKER has facilities to interpret individual CC's and to recognize the relation symbols whose value in the data base might be affected as a result of a change made at one place in the data base. CHECKER keeps track of the relation symbol, by cataloging the relation symbols in terras of their appearances in the various CC's. In general, a change in the value of one relation symbol might propogate through the data base to a series of other relation symbol values. As long as any given instance of the value of a relation symbol does not repeat itself in this series, CHECKER will have no problems. It can execute the series of necessary changes without ever having to go back to a value that it had previously changed within the sequence,

CHECKER simply performs search in the data base, and logical combinations of search. It has only simple facilities to keep track of alternate choices in search paths, and choices in possible valuations of relation symbols. Also, CHECKER can handle only constants as possible valuations for relation symbols. When the number of alternatives is large or when *loops* occur in an updating chain, the CHECKER, if left to run will keep assigning new values to the relation symbols involved until a consistent set of valuations is obtained, or until all known possibilities are exhausted. The only choices it can generate are those that are already available in the data base, or those that may be obtained by evaluating specific consistency conditions in specific local contexts. It does not have the capability to deduce logical consequences and make use of them to find contradictions where possible. To do this general theorem proving capability is necessary. The essential difference between the CHECKER and a THEOREM PROVER (TP) is the following: Whereas the CHECKER can assign as values to relation symbols only specific constants in the data base, the TP can assign as values, variables with specified logical properties. The TP can carry with it the logical properties assigned to variables and use them in making new assignments as it goes along. Resolution based theorem proving systems have this capability built into the unification algorithm [see Nilsson, 1971].

In MDS the CHECKER will invoke the TP whenever it does not find enough information in the data base to evaluate a CC at a particular anchor, or whenever the validity of an assertion is to be proven universally; not merely with respect to the facts known about the specific objects in the data base. The CHECKER will call the TP also when it recognizes a loop in an updating chain.

The deduction process and the control structure of the TP in MDS is different from that of a resolution based system, (see [10]).

2.3. The Dynamic Aspects of Modeling: The Transformation Rules and Their Interpretation

2.3.1. The Primitives

There are about twenty primitives that enable one to do programming in a backtracking environment. The primitives are classified as shown in Figure 1A. The ECP's (Environmental Control Primitives) in Figure 1A are used to establish a control environment (cenviron) within a scope. The execution of functions within the scope are affected by it. See Table III for a description of the ECP's. The SCP's are the sequential control primitives like GO, COND, etc. There are seven active primitives, GOAL, ASSERT, DELETE, CANDO, IFDON, TRY and BIND. The execution sequences for the GOAL and other active commands are shown in Figures 1B and 1C. GOAL invokes appropriate definitions from data base, and does "means-end" analysis when necessary. ASSERT and DELETE issue I and D commands to the INSTANTIATOR, when successful. All primitives, other than the control primitives, may have CANDO, IFDON and TRY functions associated with them. A primitive can be executed only if its associated CANDO's are satisfied. If a primitive fails then one may try its associated TRY functions. If a primitive is successful then its associated IFDON's should be executed. Only if the IFDON's are also successfully completed may the primitive return success to its parent. Let us follow the operation with an example.

2.3.2. Interpretation of the Active Primitives

The syntax of the various active primitives is shown in Table IV. The DESIGNER is the interpreter for the primitives. Consider, for example, the <dimension> (See syntax of <dimension> in Table IV) of the GOAL function TR1 in Table V:

```
((PEOPLE X)(PLACE P Q)(P occupants X)
←———— <bindings> ————→
    (GOAL (Q occupants X)))
    ← <fn-clause> →
```

The function call that will cause this TR1 to be invoked is

```
((PEOPLE X)(GOAL (RBANKS2 occupants X)))
<bindings> ←———<fn-clause>———→
```

Let us follow the interpretation of this function call, as specified in Figure 1B.

1) Find Possible Bindings
The CHECKER is used to bind variables in a <dimension> statement. We shall assume that the <proposition> in the GOAL-clause is always in disjunctive normal form. In the above case X will be bound to (M1 M2 M3 C1 C2 C3). If the CHECKER returns NEI, or a loop is encountered then the TP may be invoked to complete the bindings. Unless the IDB-clause (see Table III for an explanation of the IDB-clause) is present the TP will creat new objects, if necessary, to complete the bindings.

2) Find Initial Conditions
This is done by checking whether the GOAL is already satisfied in the data base for specific bindings of variables. In the case of our example, this will bring out the fact, (RBANK1 occupants (M1 M2 M3, C1 C2 C3)). This will cause the following invocation pattern to be built:

```
((PEOPLE (X1 ← (M1 M2 M3 C1 C2 C3)))
(PLACE (X2 ← RBANK1)(X3 ← RBANK2)(X2 occupants X1)
←————————— <bindings> ——————————→
(GOAL (X3 occupants X1)))←——— <fn-clause>
```

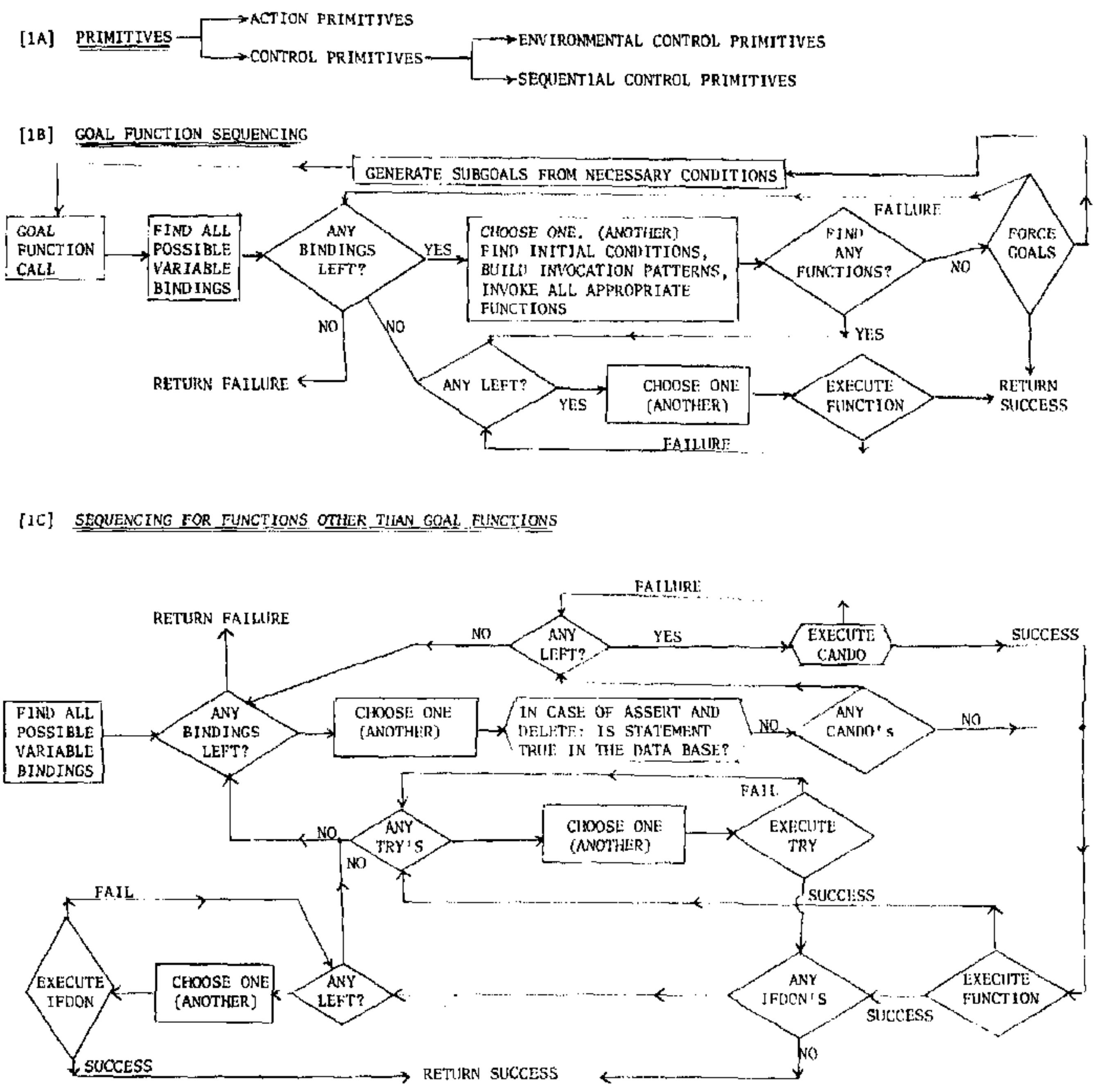

FIGURE 1: CLASSIFICATION OF PRIMITIVES AND THEIR EXECUTION SEQUENCES

Let b be the <bindings> and g the <proposition> of the <fn-clause>. The cannonical form of an invocation pattern (also a <dimension> after binding the variables) is

(<bound quantifiers>(($b_1 g_1$ v...v ($b_m g_m$)))

where each b_i and g_i is a conjunction of terms, possibly with OPNL, IFND, IDB or * clauses. Let D be an invocation pattern and D_i any <dimension> in the data base. D_i and D are said to match, ($D_i \Rightarrow D$) if there exist bindings for the variables in D_i such that for some b_{ij} in D_i and b_k in D ($b_{ij} \Rightarrow b_k$) and ($g_{ij} \rightarrow g_k$), and in addition b_{ij} is true in the data base. The b_{ij} will be the initial conditions. The invocation process will retrieve all D_i that match D.

If no such functions, D_i, are available then the DESIGNER will force the GOAL by issuing the appropriate ASSERT and DELETE commands. These will cause their associated CANDO's to be executed. If the CANDO's succeed then the corresponding I and D commands will be tried. This will cause the associated CC's to be evaluated at the given bindings of the variables, say ψ. If the CC's are not satisfied then the NTP_ψ (Not True Part of ψ) and TP_ψ (True Part) will be issued as subgoals. If the CANDO's are not satisfied then the goal will be abandoned.

In general, both the binding and invocations processes will return more than one possible course

TABLE III: THE CONTROL PRIMITIVES

	ECP'S:	ENVIRONMENTAL CONTROL PRIMITIVES
1.	SUP	SUPPRESS execution of specified class of functions within a scope.
2.	OPNL	Failure of an OPTIONAL clause will not normally cause backtracking.
3.	IFND	Backtracking can occur only if the IFNEEDED clause also failed.
4.	REPEAT	REPEAT until a given condition is satisfied.
5.	DSJN	A disjunction of statements is to hold.
6.	CNJN	A conjunction of statements is to hold.
7.	RSLE	RELEASE a previously suppressed class of functions.
8.	IDB	Bind only to objects in the data base. Do not create new objects.
9.	*	The DESIGNER cannot change *ed items; if changed their values should be restored back.
	SCP's:	**SEQUENTIAL CONTROL PRIMITIVES**
1.	GO	GO to a label.
2.	COND	Like LISP COND: backtracking under FAILURE is allowed.
3.	KILL	KILL a function and make it show either SUCCESS or FAILURE.
4.	SUSPEND	Suspend execution of a function.
5.	ACTIVATE	Activate a previously suspended function.
6.	BKTRK	Backtrack to a specified label.
	EXAMPLE:	(OPNL(REPEAT <Termination Condition>(SUP TRY)(...)(...)(ASSERT...)))

The repeat clause is in the scope of OPNL. Hence no backtracking will occur on FAILURE. Within the scope of REPEAT all executions of TRY functions are to be suppressed.

TABLE IV: THE ACTIVE PRIMITIVES: THEIR SYNTAX

```
(1)  FUNCTION DEFINITION FORMS
(A)  GOAL FUNCTIONS:
     <gfn-defn> → (<dimension><body>); <dimension> → (<bindings><fn-clause>);
     <body> → <fn-call>|<body><fn-call>; <fn-clause> → (GOAL <proposition>);
     <bindings> → <predicate>|(OPNL <predicate>)|(IFND <predicate>) |(IDB <predicate>)|<bindings><bindings>;
     <proposition> → <propositional expression which may include OPNL, IFND, IDB, and * clauses>;
(B)  CANDO and IFDON STATEMENTS and TRY FUNCTIONS
     <cifn> → CANDO | IFDON; <cifn-defn> → (<cifn><dimension><cibody>);
     <cibody> → <cistnt> | <cibody><cistnt>
     <cistnt> → (<iimension><TRY-fn>)|(<bindings><TRY-fn>)
     <TRY-fn> → (TRY <bindings><body>).
(2)  FUNCTION CALLS
     <fn-call> → <TRY-fn>|(OPNL <body>)|(IFND body)|(IDB body)|(SUP <fn-clause>)|
                 (SUSPEND)|(ACTIVATE <dimension>)|<COND-stnt>|(GO <label>)|
                 (BKTRK <label>)|<BIND-stnt> (This is like SET in LISP)|
                 <dimension>|(PROC <bindings><body> |(ASSERT <bindings>
                 <proposition>))|(DELETE <bindings><propositions>)).
```

of action. In both these cases the problem solver needs to be guided intelligently in making its choices. The DESIGNER has some built-in facilities for intelligent selection of choices from a set of alternatives. The Problem Solving State (PSS) provides this guidance. This is discussed in the next section.

2.3.3. The Problem Solving State

The PSS itself is defined by templates. The PSS template is shown in Table VI, This table is self-explanatory. Every time the DESIGNER invokes a function or executes a <fn-call> it will create an instance of PSS corresponding to the function. The network of all such PSS instances is the problem solving protocol. The CC's associated with the PSS template provide the necessary guidance to DESIGNER. Of particular interest are the CC's associated with the bindings and alternates (see Tables VI) relations. Let us call these [CCB] and [CCF], respectively. These CC's will specify the choices of current bindings and current function. Two important notions that make this possible are the notions of similarity of two PSS instances, and cc summary of a PSS instance.

cc summary: [CCS]: A CCS is a record of evaluations of CC's(branching conditions, CANDO conditions and bidding conditions, made during the tenure of a PSS instance. For each sequence of conditions evaluated, the CC-summary will contain: The TRUE RESIDUES of the conditions evaluated if the condition evaluated to TRUE, the FALSE RESIDUE, NOT TRUE PART and TRUE PART if the condition evaluated to FALSE, the RESIDUE if the condition evaluated to NEI. It will also have the outcome (fn-state) of the PSS instance in which the condition was evaluated, and specific variable bindings if any in terms of the kinds and types of objects used. All variable bindings in the CC-summary of a PSS will be specified in terms of the variables that appear in the bindings of the PSS. The concept will become clear in the example considered below. The Consistency Condition [CCB] uses CC-summaries.

The general rule is: Pick for bindings the same kind and type of objects that previously succeeded in "similar PSS instances; do not pick the kind and type of objects that previously failed. Use cc-summaries to check whether a chosen binding is likely to succeed. If no bindings could be picked by the above rules, then pick arbitrarily.

TABLE V: THE TRANSFORMATION RULES FOR THE M&C PROBLEM

(TR1) GOAL FUNCTION DEFINITION. IT SAYS, "PICK UP SOME PEOPLE Y, POSSIBLY AS MANY AS THE VEHICLE WILL HOLD, LOAD THE VEHICLE WITH Y, AND TAKE THEM TO Q". "PROC" IN THE STATEMENT BELOW STANDS FOR PROCEDURE. IT IS SIMILAR TO PROG IN LISP, BUT DIFFERS FROM PROG IN THE SENSE THAT PROC HAS BACKTRACKING.

```
(((PEOPLE X)(PLACE P Q)(P occupants X)(GOAL(Q occupants X))
 (REPEAT (Q occupants X)
  (PROC ((SOME VEHICLE V)(SOME PEOPLE Y)(OPNL(Y #.is.capacity of V))(Y among X))
        1.  (ASSERT (V occupants Y)(NOT(P occupants Y)))
        2.  (ASSERT (V position Q)(NOT(V position P)))
        3.  (ASSERT (V occupants.are.occupants of Q)(NOT(V occupants Y))))))
```

(TR2) CANDO CLAUSE FOR LOADING A VEHICLE. IT SAYS, "UNLOAD THE VEHICLE FIRST. IF THERE ARE MORE PEOPLE THAN THE VEHICLE CAN HOLD THEN DROP SOME AND TRY AGAIN. IF THE VEHICLE IS NOT AT THE SAME PLACE AS THE PEOPLE ARE, THEN BRING IT TO WHERE THEY ARE. *EACH TRY STATEMENT IS PRECEDED BY THE CONDITIONS, ON* WHOSE FAILURE THE TRY WOULD BE ATTEMPTED.

```
(CANDO ((PEOPLE X)(PLACE P)(VEHICLE V)(P occupantsX)(ASSERT(V occupants X)(NOT(P occupants X)))
        1.  (ASSERT(V position P))
        2.  ((PEOPLE Z)(V occupants Z)(ASSERT(P occupants Z)(NOT(V occupants Z))))
        3.  ((X #.≤.capacity of V)
             (TRY ((SOME PEOPLE Y)(Y among X)(ASSERT(V occupants Y)(NOT(P occupants Y))))
                  (IFDON(KILL(ASSERT(V occupants X)(NOT(P occupants X))))))))
```

(TR3) CANDO CLAUSE ASSOCIATED WITH BRINGING A VEHICLE FROM A PLACE P TO Q. IT SAYS, "GET A PILOT AND TAKE THE VEHICLE. IF PILOT CANNOT BE REMOVED GET SOME ONE TO GO WITH HIM".

```
(CANDO ((VEHICLE V)(PLACE P Q)(V positionP)(ASSERT(V position Q)(NOT(V positionP))))
        1.  (((SOME PERSON Y)(V pilot Y)(V occupant Y))
            1A. (TRY((SOME PERSON Z)(V pilot Z)(P occupants Z)
                                   (ASSERT (V occupants Z)(NOT(P occupantsZ)))))
            1B. (TRY((SOME PERSON X Y)((V pilot X)v(V pilot Y))(P occupants (X Y))
                     (ASSERT(V occupants(X Y))(NOT(P occupants (X Y)))))))
```

TABLE VI: PSS TEMPLATE

(dimension	DIMN)	: <dimension> of the function
(intl. state	MODEL STATE)	: Initial State of Model
(bindings	BINDINGS)	: All possible variable bindings and current bindings
(alternates	PSSL)	: PSS instances of possible alternates; also the currently active function
(cc-summary	CCSL)	: *summary of CC's evaluated during the active tenure of fn*
(fn-state	FNSTATE)	: Function States: ACTIVE; SUCCESS, FAILURE, SUSPENDED
(cenviron	CENVIRON)	: The Control environment
(history	PSSL)	: Previous instances of PSS with the same dimension
(type	PSSTYP)	: Any useful type classification of PSS
(final-state	MODELSTATE)	: Model changes done by the PSS instance
(successor	PSSL)	: Possible successor functions
(predecessor	PSSL)	: List of parent functions
(conditions	CONDN)	: Conditions appearing in REPEAT, COND and other such statements that caused *the current PSS to be invoked*

To define the notion of similarity of two PSS states we need some additional concepts. Let k be an arbitrary PSS instance defined as follows:

k: (dimension D_k)(bindings B_k)(initl-state I_k)
(alternates A_k)(fn-state S_k)(cenvirons E_k)
(cc-summary CCS_k)(history H_k)(Type T_k)
(final-state FS_k*)(successor* U_k*)(predecessor* R_k*)*
(conditions C_k)

Let $V_k = \{X_1, X_2, \ldots, X_n\}$ be the variables appearing in the <dimension>, D_k. Let D_k itself be $(Q_k P_k F_k)$ where Q_k is the <quantifiers> of D_k, P_k its <proposition> and F_k its <fn-clause> (see Syntax of <dimension> in Table IV). Let $\beta_k = \{(X_i \leftarrow A_i) \mid 1 \le i \le m\}$ be a specific binding of the variables in V_k such that $\beta_k(P_k)$ is TRUE. The bindings B_k is the set of all such β_k. The initial-state, I_k, is a conjunction of terms defined on the variables in V_k and possibly some constants in the data base. Let $N_k = \{N_1, N_2, \ldots, N_n\}$ be the constants in I_k. In each PSS instance k, I_k^n is the same as the final-state of its predecessor, together with whatever might have been added by the function invocation process performed in k, all expressed in terms of the variables in V_k and constants in N_k. For every $\beta_k \in B_k$ $\beta_k(I_k)$ is true for the given constants in N_k. Not all of the constants in N_k might be used in the PSS *instance. The used ones are precisely those that* appear in the CC-summary of the PSS instance. Let N_k^o denote the used ones in *k*.

Two PSS instances K and J are similar if
(i) They have the same dimension, $D_k = D_J$. Thus $V_K = V_J$, and K and J are in the same history list, $H_K = H_J$. Also, the type of K is equal to the type of J.
(ii) They have the same control environment,
$E_k = E_j$, and

(iii) V_K,N_K, satisfy one or more of the cc-summaries in J, for some binding β_k ϵ B_k. So also, V_J,N_J satisfy one or more cc-summaries in K for some binding β_J ϵ B_J.

K is identical to J if D_K=D_J, T_K=T_J, E_K=E_J, B_K=B_J, and N^Δ_K = N^Δ_J.

To understand how these work let us consider the solution of the M&C problem.

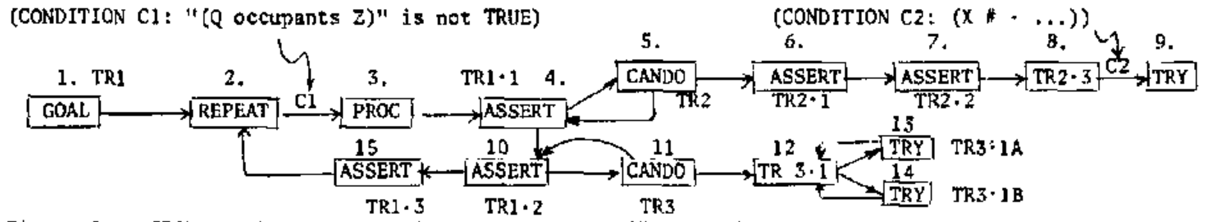

Figure 2: GRAPH OF FUNCTION CALLS IN THE SOLUTION OF THE M & C PROBLEM.

2.3.4 M&C Problem Solution

The sequence of possible function calls is shown in Figure 2. Each box in Figure 2 is labelled to indicate its correspondence to the functions in Table V. The boxes are numbered 1 through 15. For a box with number i, let K_i denote its associated PSS instance.

Suppose we are at the beginning, and are at box 3 in Figure 1. Then the following sequence of actions might happen (follow arrows in order):

$$\begin{tikzcd}
(Y \leftarrow (M1\ M2)) \arrow[r, "1"] & \text{enter box 4} \arrow[dl, "4"] \arrow[d, "2"] \\
I(V\ \underline{\text{occupants}}\ Y) & \text{enter box 5} \arrow[u, bend right, "3"'] \\
D(P\ \underline{\text{occupants}}\ Y) &
\end{tikzcd}$$

The indicated I and D commands are returned by the ASSERT function, TR1·1 (see Table V). This will cause [CC4] and [CC1] to be evaluated, and the following cc-summaries to be returned to box 3 (since box 3 is still active)*:

[CCS4][K_4, I(V occupants Y), (V ← VEHICLE) (Y ←(M,M)),
T, Fail]
[CCS1][K_4, D(P occupants Y), (P ← PLACE) (Y ← (M,M)),
F, Fail]

Here K_4 is the PSS instance at which the evaluation took place. The variable bindings are indicated only in terms of the kinds and types of objects used. T and F are the CC evaluation outcomes, and Fail is the outcome of K_4. Notice that moving these two cc-summaries to K_3 makes it still possible for K_3 to use these, because the variables P and V have in K_4 the same bindings as they do in K_3 and none of the terms appearing in the CC's have changed in value between K_3 and K_4.

The failure of K_4 brings us back to K_3. Now the choice of next bindings to be tried will be guided by [CCB]. Either a (M,C) or a (C,C) will succeed. A more interesting case is the following.

* We shall use M for Missionaries, C for Cannibals, Fail for Failure, T for True, SUC for success, and F for False.

Now suppose that (M1,C1) are already on RBANK2. This would have caused the following series of successful cc evaluations:

[CCS4']:[K'_4 , I(V occupants Y), (V ← VEHICLE)
(Y ← (M,C)). T, SUC]
[CCS1']:[K'_4,D(P occupants Y), (P ← PLACE) (Y ← (M,C)),
T, SUC]
[CCS2]:[K_{10}, I(V position Q), (Q ← PLACE)
(V ← VEHICLE), T, SUC]
[CCS1'']:[K_{15}, I(Z occupants of Q), (Q ← PLACE),
(Z ← (V occupants ?)) (V ← VEHICLE),T,SUC]

All these cc-summaries would now be available at box 3 of Figure 2, since PROC *would have been still active* during the whole course of events.

After this, PROC will be reinvoked because REPEAT will have been still active. The new instances of PSS for boxes 3 and 4 will be created. These will be similar to the previously created instances. The BOAT is now at RBANK2. The CANDO clause, box 5, (statement TR2 in Table V) will now cause the BOAT to be brought back to RBANK1 with a pilot, who in this case will have been the missionary M1. This time, when a new pair of PEOPLE are picked from RBANK1, the system will already check for the satisfaction of the successful path of CC executions, depicted by the summaries [CCS4'], [CCS1'], [CCS2], [CCS1'']. Picking another (M,C) will in this case fail; (C,C) will succeed in satisfying the cc-summaries. Thus, with anticipation the system will pick the right candidates likely to lead to success. From here on the availability of the cc-summaries, and the guidance provided by [CCB] will enable the system to always pick the right candidates. The following solution will be obtained:

	RBANK1	RBANK2
1	(M1,M2,M3C1,C2,C3)	none
2	(M2,M3,C2,C3)	(M1,C1)
3	(M1,M2,M3)	(C1,C2,C3)
4	(C1,M3)	(M1,M2,C2,C3)
5	(C1,M3,M1,C2)	(M2,C3)
6	(C1,C2)	(M1,M2,M3,C3)
7	(C1,C2,C3)	(M1,M2,M3)
8	(C1)	(C2,C3,M1,M2,M3)
9	(C1,C2)	(C3,M1,M2,M3)
10	none	(M1,M2,M3,C1,C2,C3)

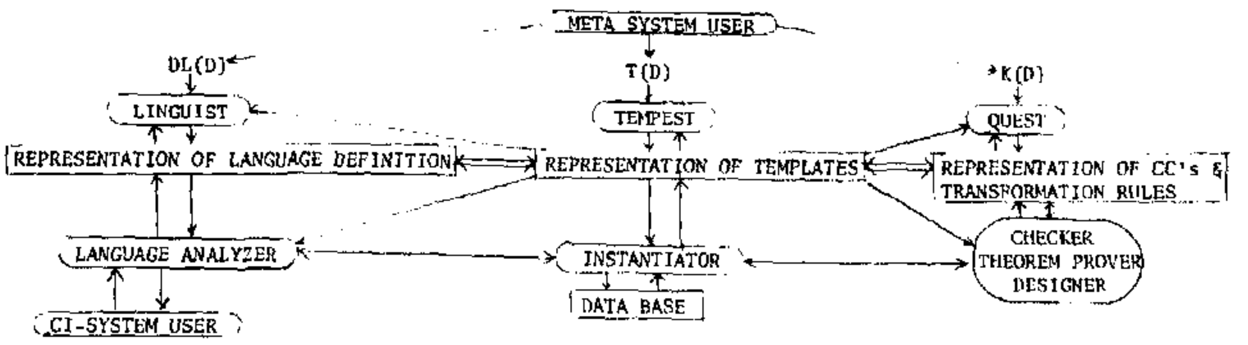
META SYSTEM USER
DL(D)
T(D)
K(D)
LINGUIST
TEMPEST
QUEST
REPRESENTATION OF LANGUAGE DEFINITION
REPRESENTATION OF TEMPLATES
REPRESENTATION OF CC's & TRANSFORMATION RULES
LANGUAGE ANALYZER
INSTANTIATOR
CHECKER THEOREM PROVER DESIGNER
DATA BASE
CI-SYSTEM USER

FIGURE 3: Block diagram of MDS: " <→" indicate pointers in data representations; "←→" indicate data and control flow paths; ⊏⊐ denote data items and (⊂⊃ denote processors.

Step (5) in the above solution is caused by box 14 in figure 2.

2.3.5. Summary

Thus, the DESIGNER provides the high level control structure necessary to pass on to the CHECKER the right CC's to be evaluated, and to the INSTANT1A-TOR, the right model changes to be done. The DE-SIGNER programs themselves are independent of the descriptive data structures used. Again the templates and INSTANTIATOR provide a desirable isolation. The PSS itself may be changed for different domains of discourse, or different problem types. In this sense, *the* templates and the rules of transformation, to-gether with the PSS specialize the MDS to a given problem, or a given domain of discourse. The problem solving control structures are driven by the domain dependent data. The CHECKER, TP, DESIGNER, anri IN-STANTIATOR are alt part of the MDS.

Most importantly there is a significant strat-ification of knowledge in a domain, as seen by the system. Domain dependent knowledge is made available to the system as templates, as CC's or as TR's, The PSS templates play a particularly important role. Depending upon how and where a given piece of domain dependent knowledge is specified the system uses it differently.

The relative isolation of the problem solving and model management programs from the descriptive data structures themselves, make the concept of MDS feasible. The facility to arbitrarily specify des-criptive data structures as well as non-deterministic programs makes the system highly flexible and power-ful. The CHECKER and INSTANTIATOR provide the basic foundation. These two systems are small systems (about 2K PDP-10 words for INSTANTIATOR and 3X for CHECKER), and the programs here can be made very efficient. These features give promise that the proposed system architecture could operate in the context of large data bases. By defining the tem-plates carefully the MDS system can be specialized to operate efficiently in a given domain. The structure of MDS is described in the next section.

3. The Meta Description System

The block diagram of MDS is shown in Figure 3. In this figure DL{D), T(D), and K(D) are, respectively, the definitions of Descriptive Language, Templates and Knowledge (CC's and TR's) in a domain D, The LINGUIST, TEMPEST, and QUEST are, respectively, the subsystems that accept these definitions and create representations for them. The TEMPEST is now a working system (about Sk of PDP-10 words of compiled LISP 1.6 programs). The CHECKER and INSTANTIATOR are presently under construction.

The data in DLp), T(D), and K(D) specialize the MDS for the domain. The rest of the block diagram is self explanatory.

4. Concluding Remarks

We have introduced the basic concepts *of C1-* Systems and the MI'S. The CI-Systems provide a basis for the definition of the concept of machine under-standing in terms of models that a machine is capable of building in a domain, and the way the models are used. The understanding exhibited at the problem solving, level of CHECKER is relatively simple under-standing. A deeper level of understanding is exhibited in the kinds of problems that the Theorem Prover can solve (see [in]). At the level of DESIGNER the level of understanding is very sophisticated. The system is able to plan and build procedures to solve problems.

In this paper we have discussed only a part of the problem solving aspects *of* the system; the workings of the CHECKER and DESIGNER. The operation of the langu-age processor will be discussed in a subsequent paper.

We are proposing the use of DL(D), T(D), and K(D) to transfer domain dependent descriptive knowledge to a computer. We have briefly indicated how such des-criptive knowledge could be used to solve problems in a domain automatically.

The specification of *DL(D1,* T(D) and K(D) in a domain will, of course, require a very good under-standing of the concepts and problems in a domain. There are several domains where, at present, such understanding *is* available. The MDS provides a way of transfering this understanding to a computer. The study of a CIS for the MDS itself might throw light on the problem of making *a* computer build its own tenjplates to suitably model and reorganize a known corpus of knowledge in a domain.

There is much work to be done to make the MDS a viable system. It is necessary to develop a working system first. We are presently involved in this task.

References:

1.) Fikes, Richard Earl, "REF-ARF: A System for Solving Problems Stated as Procedures," J. Art. Intel. 1(1) 1970.

2.) Fikes, Richard Earl, "A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures," Doctoral Thesis, Carnegie-Mellon University, 1968.

3.) Derksen, J., Rulifson, J.F. and fValdinger, R.J., "The QA4 Language Applied to Robot Planning," AFIPS Conference Proceedings, Vol. 41, Part II FJCC 1972, pp. 1181-1187.

4.) Gibbons, Gregory Dean, "Beyond REF-ARF: Toward an Intelligent Processor for a Nondeterministic Programming Language," Doctoral Thesis, Carnegie-Mellon University, 1973.

5.) Fikes, Richard Earl, Nilsson, Nils J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," J. Art. Intel. 3(1) pp. 27-68, 1972.

6.) Fikes, R.E. , Hart, Nilsson, N.J. : "Learning and executing generalized Robot Plans", J. Art. Intel. 3(1972), 251-288.

7.) Hewitt, C, "Description and Theoretical Analysis (using schemata} of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. Thesis, Dept. of Mathematics, M.I.T., Cambridge, Mass. 1972.

8.) Newell, A., Shaw, J.D., and Simon, H.A., "Report on a General Problem-Solving Program for a Computer," Information Processing: Proc. Internl. Conf. Information Processing, p. 256-264, UNESCO, Paris. (Reprinted in Computers and Automation, July 1959)

9.) Amarel, S., "On Representations of Problems of Reasoning About Actions," Machine Intelligence 3 D. Michie, ed., Edinburgh University Press, pp. 131-170. 1968.

10.) Srinivasan, C.V., "On the Organization and use of Knowledge in a Coherent Information System" RUCBM-TR19. Dept. of Computer Science, Rutgers University, New Brunswick, N,J, 08903,