

S-NET : A FOUNDATION FOR KNOWLEDGE
REPRESENTATION LANGUAGES

Makoto Nagao
Department of Electrical Engineering
Faculty of Engineering
Kyoto University
Kyoto, JAPAN

Jun-ichi Tsujii
Department of Electrical Engineering
Faculty of Engineering
Kyoto University
Kyoto, JAPAN

A new knowledge representation scheme called S-Net is presented. The S-Net is a descendent of both semantic networks and recently developed AI languages. We are willing to introduce procedures into our network notations as FRL and KRL do. However, these languages have a serious disadvantage that the programmer should specify, explicitly by using indicators such as WHEN-FILLED, TO-FILL etc., when the attached procedures are invoked. This results in the restriction of system's abilities of solving problems. To avoid this, the programmer should always pay attention to the overall control issues during the coding of his procedures. In our new formalism, the explicit specification is not necessary. The problem solver based on the S-Net dynamically determines which procedure is invoked when, according to the problem solving situations, not to the pre-specified indicators. We also discuss the problems of property inheritance through hierarchy. The inheritance mechanism provided so far is so restrictive that the programmer should do all things in his procedures. In the S-Net, the property inheritance is further augmented by the 'explicit path specification'. The detailed construction of the problem solver, which performs both forward and backward reasonings appropriately, is also given.

1. INTRODUCTION

In this paper, we discuss a graphical notation called S-Net, which results in a foundation for recently developed AI languages. S-Net is a descendant of both semantic networks and recently developed AI languages. Semantic networks have been developed since the mid-sixties as a formalism for the representation of knowledge. Though the original idea of semantic networks is intuitively attractive, the recent formalisms of networks seem to lose the advantages which the researchers initially conceived. G. Hendrix [1975], L. Schubert [1976] and others have tried to formalize their networks based on predicate logic. As a result, their notations have become graphical analogues of logical formula. On the other hand, since the appearance of 'Frame' [Minsky 1975], great efforts have been made to develop programming languages and systems which realize the frame idea [Winograd 1977, Bobrow 1976, Goldstein 1977, Davis 1978]. They all have a certain data structure into which various sorts of knowledge, including both procedural and declarative ones, are integrated. However, their data structures are based only on the vague idea 'Frame', and, more seriously, lack rigidly

defined semantics. Especially, the technique 'procedural attachment', though it is powerful tool for describing knowledge, makes the situation chaotic. There are no general mechanisms for managing the attached procedures appropriately. In most languages, the programmer must specify explicitly when an attached procedure should be invoked. Moreover, most aspects concerning the control of the interactions among the attached procedures are also left to the programmer. As a result, the programmer must always pay attention to the overall control issues during the coding of his procedures. We present in this paper a framework called S-Net on which descriptions and operations of these languages can be appropriately founded.

2. RELATED RESEARCH WORKS

Among others, FRL [Goldstein 1978] and KRL [Bobrow 1977] are the better known examples of recent AI languages. Because these two have both the advantages and disadvantages of recent attempts, we will consider them as typical examples and compare them with the S-Net notations throughout this paper.

As Goldstein said, FRL is a programming system rather than a language. It lacks the overall control mechanisms. It only provides a data structure and a programming package for manipulating it. The data structure, a frame system, consists of a set of mutually related knowledge units called 'frame'. A frame, in turn, is an aggregation of descriptive data and procedures. The core of a FRL frame is a property list. FRL generalizes the LISP property list in the following ways:

1. FRL supplies a mechanism for inheritance of properties through a generalization hierarchy.
2. The property value in a frame can be a procedure which knows how to get the actual value.
3. Procedures can be associated directly with frames. The programmer is required to specify explicitly, by using facet indicators such as \$IF-ADDED, \$IF-REMOVED, \$IF-NEEDED etc., what sorts of operations on the frames will invoke the associated procedures.

In short, the essential features of FRL are

1. Inheritance of properties through a hierarchy.

2. Procedural attachment.

Though these two ideas are intuitively attractive, there remain a lot of difficult problems before they can be realized. We will consider these problems in this section. Note that because we restrict the discussion in this paper to the topic of problem solving mechanisms and their foundations, we will neglect many characteristic features of FRL or KRL which are very interesting in their own right.

2.1 Property Inheritance Through a Hierarchy

Fig. 1 shows an example in [Goldstein 1977]. The frame MINSKY inherits the properties STREET, CITY, STATE and ZIP from the frame MIT-AI. In FRL, 'the AKO (A Kind OF) link establishes a mapping by which properties of one frame may be distributed to related frames. Semantically, the mapping can represent a set/subset, part/whole, or other relationships [Goldstein 1977].

```
(FASSERT   MIT-AI
 (STREET ($VALUE ( /545 Technology Square/ )))
 (CITY   ($VALUE ( /Cambridge/ )))
 (STATE  ($VALUE ( /MA/ )))
 (ZIP    ($VALUE ( /02139/ ))) )

(FASSERT   MINSKY
 (AKO      ($VALUE ( MIT-AI )))
 (OFFICE   ($VALUE ( 821 ))) )
```

Fig. 1 Description of an AKO Hierarchy in FRL.

This approach seems very promising and appropriate. However, some properties can be inherited through set/subset relations but not through part/whole relations and vice versa. From the view point of logic, property inheritance is a specific mode of inference associated with a predicate. In an extreme case, for example, children may inherit their second names from their parents but not any other properties. This inheritance rule is associated with a predicate, say, CHILD-OF. In FRL, there are no explicit provisions for representing such a kind of restricted property inheritance. The only possible solution is to embed such a rule in a procedure (Fig. 2). This solution is not a real solution of the problem. The programmer should do all things in his procedures. Because property inheritance through hierarchy provides only a very restricted inference mechanism, we need additional frameworks for supporting other kinds of inheritance mechanisms.

```
(FASSERT   HUMAN
 (SECOND-NAME ($VALUE (! (Get-Name) )))
```

This procedure will search the parent of the current Human and get their second name.

Fig. 2. Restricted Property Inheritance Embedded in a Procedure.

2.2 Procedural Attachment

One of the difficult problems in procedural attachment is how to decide when a procedure is evaluated. In most of the AI languages, the programmer should explicitly specify this.

```
DAY Frame
Year
Month
Day-Number      WHEN-FILLED
                  (CHECK-RELATION day-number, month)
Day-of-Week      TO-FILL
                  (APPLY calendar-lookup
                    TO year, month, day-number)
                  (APPLY anchor-date-method
                    TO year, month, day-number)
ASCII-form*      WHEN-FILLED
                  (FILL year, month, day-number)
```

* ASCII-form for July 4, 1978 is 780704.

Fig. 3. Description for 'Day' in KRL.

\$IF-ADDED in FRL and WHEN-FILLED in KRL indicators are used to indicate that the attached procedure should be invoked when a certain value is stored in the slot. However, this style of

specification is often inadequate. Some procedures should be triggered when more than one slot is filled. Fig. 3 is an example by Winograd [1975]. The procedure Check-Relation, which checks the consistency between the Day-Number and the Month, will be triggered when the Day-Number slot is filled. If the Month slot has not been filled when the Day-Number is filled, the execution of the procedure should be suspended, and resumed again when the Month slot is filled. The control of such suspension and resuming of procedures may become very complicated, if it is implemented without any clear formalization. Moreover, the specification of TO-FILL and WHEN-FILLED indicators results in the restriction of the system's abilities. Consider a simple example in which we have two procedures which compute the slot-B's value from the slot-A's value, and the slot-C's value from slot-B's value respectively. There are four possibilities to embed such procedures by using TO-FILL and WHEN-FILLED indicators [Fig. 4]. In the first three, the slot-C's value can be obtained when the slot-A's value is given. However, it is not the case in the last representation.

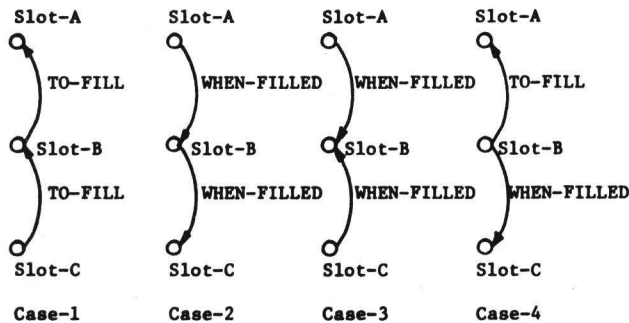


Fig.4 Explicit Pre-specifications of Invocation

To avoid this, we should carefully distinguish the two different kinds of knowledge: what logical or conceptual relationships the attached procedure embodies, and when the procedure will be invoked. It is often known beforehand that the evaluation of a certain procedure requires much more resources (time or space) than the others or that a certain direction of searching often leads us to success. These sorts of knowledge are called 'heuristics' and the programmers have been encouraged to embed them in their representation. However, it is our contention that such heuristics are only heuristics, i.e., they may be reflected on the problem solving process only to the extent that they will not restrict the system's abilities. TO-FILL (WHEN-FILLED) indicators not only recommend the backward (forward) usages of the attached procedures but also inhibit their forward (backward) usages.

In short, the pre-specification of the usages of attached procedures is too restrictive.

Our problem solver based on the S-Net alternately performs backward and forward reasonings. In that process, an attached procedure will be utilized either in forwards or backwards, according to the problem solving situations. In order to accomplish such mechanism, we introduce the concept of 'partially instantiated relationships' in the S-Net, which are dynamically created during the problem solving process [See 4.1 and 4.2].

3. S-NET REPRESENTATION

S-Net consists of several types of nodes and links. The node types which are used in the S-Net are shown in Fig. 5. The Disjoint node and the Condition node play the role of logical connectives. They are used for augmenting the generalization hierarchy (3.1). A property or a slot is represented in the S-Net by a node called Property node. All variable nodes in the S-Net are typed by certain data types. The definition of a type plays a central role in the S-Net.

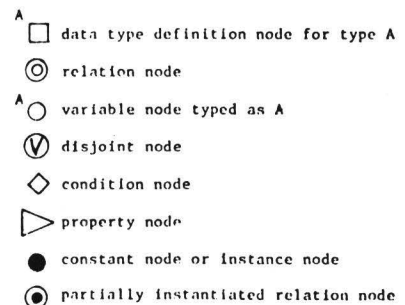


Fig. 5. Node Types in the S-Net.

A Type Definition node describes the basic behaviour of a data type. It corresponds to an atom in LISP. In LISP, an arbitrary character string identifies an atom, and all occurrences of the same character string in LISP programs and data are represented internally by a pointer which points to the atom. The behaviour of an atom is described by the property list attached to the atom. According to this convention, a data type in the S-Net is identified uniquely by its name, and there is one type definition node for each data type.

3.1 Augmentation of Generalization Hierarchy

Data types can be organized in a hierarchical structure. We use the term SORT, instead of

AKO or ISA, to refer to the link. In our problem solving procedure, the hierarchy of data types plays the central role for distributing properties, though explicit path specification (See 3.3) will further augment the inheritance mechanisms. When a problem description is given, the instance nodes of the corresponding data types are created. Because each instance in a problem description has its own data type name, the initial position of the instance in the hierarchy is determined by its name. However, the position should be determined not only by its type name but also its whole descriptions. For example, the instance

```
(OBJECT-1  NAME = ALCOHOL
      (! TEMPARATURE) = 28°C )
```

is identified with the data type ALCOHOL by its type name. Moreover, because ALCOHOL is a sub-sort of MATERIAL and because an instance of MATERIAL whose temperature is between the boiling point and the congelation point is an instance of LIQUID, it should be identified as an instance of LIQUID. All properties of LIQUID should be distributed to this instance. In order to accomplish such an identification process, we need a framework in the S-Net for specifying when an instance is located lower in the hierarchy. For this purpose we introduce in the S-Net a new link SUBSORT and a new type of node Condition Node (See Fig. 6). A SUBSORT link is a link from a higher data type to a lower one. A Condition Node holds a condition when the transition from the higher to the lower data type is permitted. Moreover, because new information about an instance may be derived during the problem solving process or accumulated through the dialogue with the user, the position identification may be performed during the problem solving process.

We often find the following type of declarative sentences.

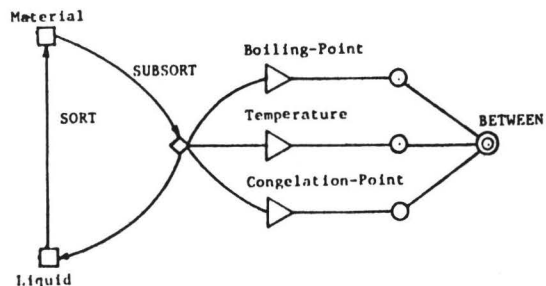


Fig. 6. SUBSORT Links with Condition Nodes.

Acid which dissolves copper is either Hydrochloric-Acid, Nitric-Acid or Sulphuric-Acid.

These statements describe mutually exclusive relationships among sets. A data type and a set do not necessarily correspond but they are closely related.

We call a set of data types 'Disjoint Set', if the data types in the set satisfy the following conditions:

1. If an instance is a certain data type in the set, it can never be other data types in the set.
2. We can determine the data type of an instance by determining which data type it is not equal to, i.e. by elimination.

In the S-Net, we represent a disjoint set by a node called Disjoint Node (Fig. 7). Disjoint nodes are often useful to inhibit semantically irrelevant information from entering into the search space. By combining the Condition nodes and Disjoint nodes, the generalization hierarchy in the S-Net becomes very powerful.

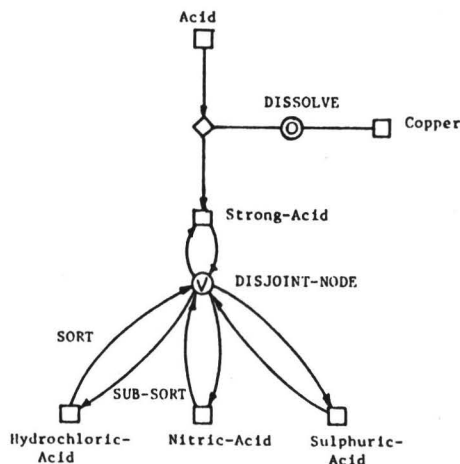


Fig. 7. DISJOINT-NODE.

3.2 Relation Nodes

A relation node in the S-Net expresses a certain dependency relationship among the slots, and also functions as entrance to some external procedures from the S-Net. Based on the dependency relationships, the S-Net problem solver determines how to solve a problem. For each relation node, a bunch of procedures is internally defined. The procedures embody how to utilize the relation in the problem solving. The problem solver decides which internal procedures is invoked,

based on both the dependency relationships and the problem solving situations.

Logical predicates of certain kinds, arithmetic operators, external data bases and procedures in general are all equally represented in the S-Net by relation nodes. A relation node essentially corresponds to a logical predicate. A logical predicate extensionally specifies a set, possibly infinite, of tuples which satisfy the predicate. We assume that external data bases are constructed according to the relational data model. It consists of the tuples which satisfy a predicate or relationship, while a procedure in general expresses the mechanism for computing those tuples.

The description of a procedure in a relation node consists of two parts: Declaration part and Procedure body. The declaration part shows in which direction the procedure utilizes the relationship. This part, as demonstrated below, is fairly simple:

$$(arg_{i_1} \ arg_{i_2} \ \dots \ arg_{i_n}) : (arg_{i_{n+1}} \ \dots \ arg_{i_m})$$

This means that if $arg_{i_{n+1}}, \dots, arg_{i_m}$ are known, $(arg_{i_1}, \dots, arg_{i_n})$ can be computed by the procedure. The declaration part can be interpreted both forward and backward by the S-Net problem solver depending on the situation, i.e. if one of $arg_{i_1}, \dots, arg_{i_n}$ is required, then try to get the values of $arg_{i_{n+1}}, \dots, arg_{i_m}$ (in the backward mode), or if $arg_{i_{n+1}}, \dots, arg_{i_m}$ are all known, then try to compute the values of $arg_{i_1}, \dots, arg_{i_n}$ by the procedure (in the forward mode). As for the backward and forward mode, see 4.2.

In a special case, the left hand side of the declaration part can be null. In this case, the procedure body will be invoked when the arguments of the right hand side are all filled, and check the consistency among them.

Some examples of relation nodes are shown in Fig. 8.

3.3 Nesting of Data Types — Explicit Path Specification

A data type or a frame can be recursively defined. A data type may arise in the definition of another type. The description

(DT NAME = Aqueous - Solution

(! Solvent) = (DT NAME = Water))

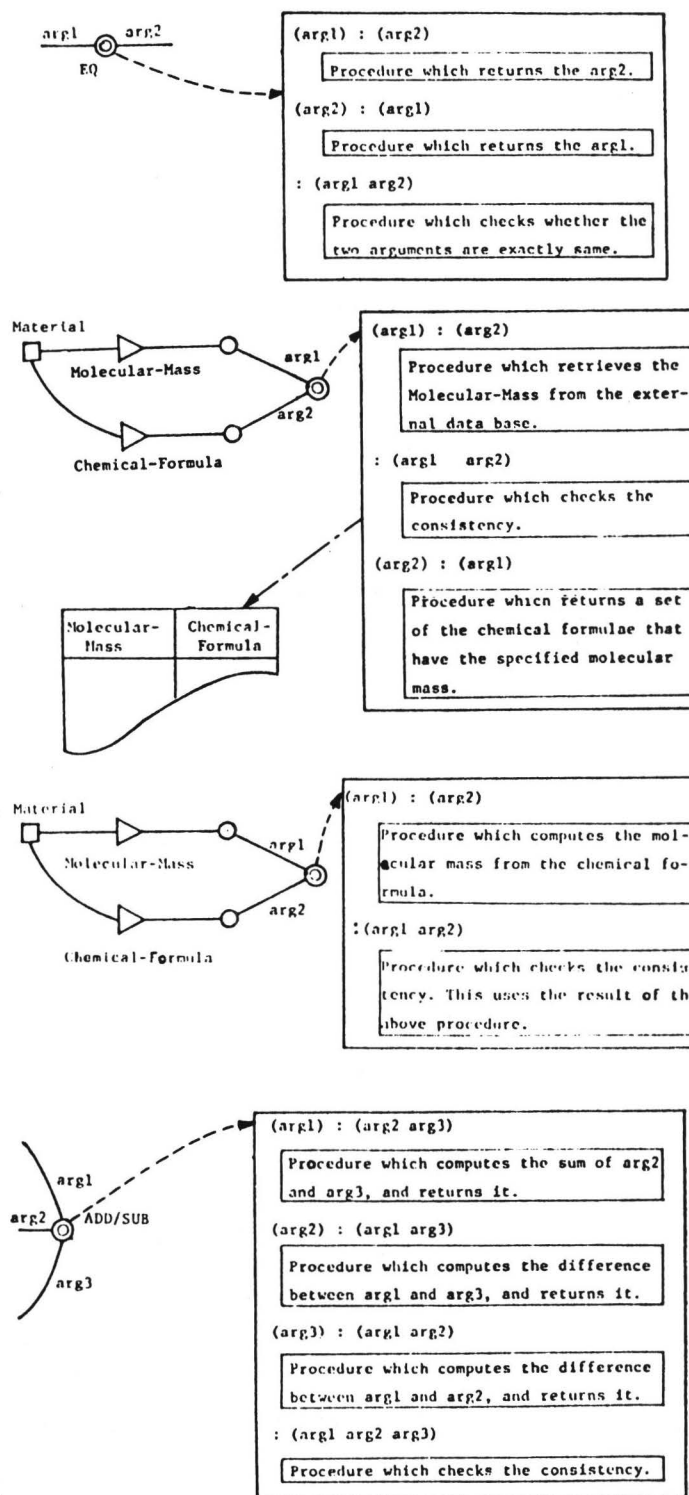


Fig.8 Examples of Relation Nodes and Their Internal Procedures

is a simple example of such descriptions. We treat an occurrence of a data type in the definition of another type as a variable typed by the data type. Thus, (DT NAME = Water) in the above example is represented as a typed variable. Since the variable is typed as Water, all information attached to Water is applicable to the variable. For example, we will have the value H2O when we retrieve the value of the property Chemical-Formula of the Solvent of a certain Aqueous-Solution. Moreover, a specific knowledge, which can be applied only to the water that is the solvent of a certain solution, but not to Water in general, is also attached to the variable.

```
(DT NAME = Aqueous - Solution
(! Solvent) = (DT NAME = Water)
(ADD/SUB (! MASS) (! Solvent MASS)
(! Solute Mass)))
```

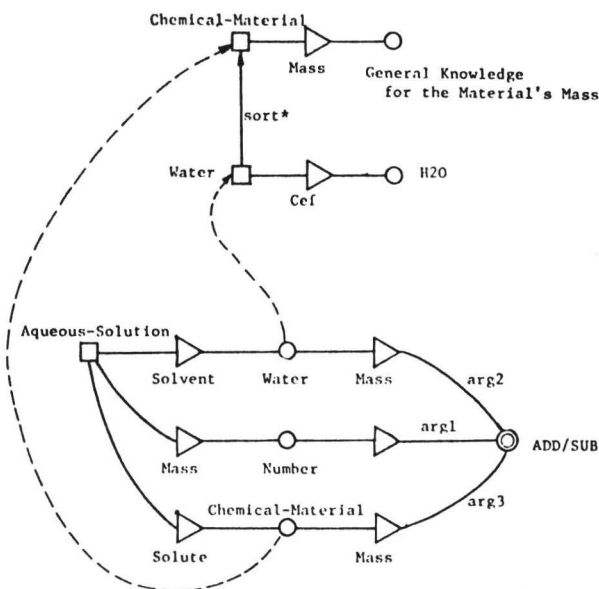


Fig. 9. Type Definition Node for 'Aqueous-Solutions'.

The S-Net for the above description is shown in Fig. 9. Retrieval of a value through a path description such as (FGET Aqueous - Solution Solvent Mass) will cause the evaluation of a procedure which is defined as the 'internal' procedure of the relation node ADD/SUB that computes the Solvent's mass from those of the Solution and the Solute. Moreover, if the procedure fails to compute it, more general procedures which are attached to the data type

Water or more general data types than Water will be invoked. Thus, unlike in FRL, a path description in the S-Net is augmented automatically by the system.

Such a path description can be employed not only in the query expression but also in the S-Net descriptions as follows. The general statement

'If a researcher is a member of an organization, he inherits the address from the organization'

can be expressed by

```
(DT NAME = Researcher
(! Organization) = (DT NAME
= Organization)
(EQ (! Address) (! Organization
Address))).
```

In the above examples, all path descriptions begin with !'s which show that the paths start from the current data type. A path which starts from a data type that is different from the current one is also allowed in the S-Net. Thus, the statement 'A MIT-Researcher inherits his address from MIT-Al-Lab' can be expressed directly by

```
(DT NAME = MIT - Researcher
(EQ (! Address) ((DT NAME =
MIT-Al-Lab) Address))).
```

The S-Net for this is shown in Fig. 10. By using this mechanism, called 'explicit path specification', property inheritance or distribution among different data types can be obtained in a very intuitive way. Because EQ is represented as a relation node, the above expression can be used not only for retrieving the researcher's address but also for checking the consistency when the address of a certain MIT-Researcher is given.

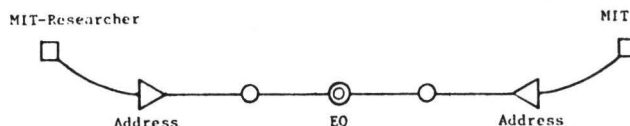


Fig. 10. Related Type Definition Nodes.

4. OPERATIONS ON THE S-NET

4.1 Creation of an Instance

A problem description may contain several instances. The description

```

(OBJECT-1  NAME = Solution
  (! Mass) = 100 gr
  (! Solvent) = (OBJECT-2  NAME = Water
    (! Mass) = 98 gr)
  (! Solute) = (OBJECT-3
    (! Cef) = Na-Cl))

```

contains three instances of different data types. They are instances of Solution(OBJECT-1), Water (OBJECT-2) and a data type without a name(OBJECT-3).

The type definition node for Solution contains various relationships among the properties. These relationships are also instantiated. For example, there may exist a relation node ADD/SUB that expresses the relationship among Solution's mass, Solvent's mass and Solute's mass. When the solution's mass is filled by 100 gr, the relationship is also partially instantiated, and the partially instantiated relation node is created (Fig. 11).

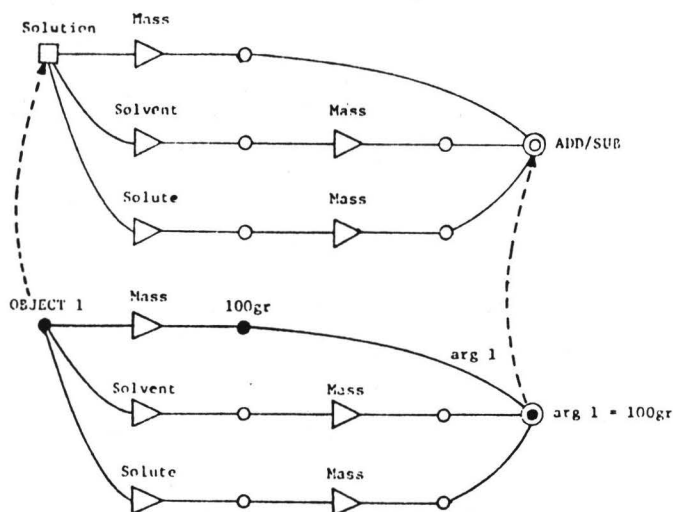


Fig. 11. Partially Instantiated Relation Node.

The instance OBJECT-2 is created as an instance of Water. Because the property Mass is filled by 98 gr, the relation nodes attached to Water's Mass are instantiated. Moreover, because OBJECT-2 is put in the property Solvent of OBJECT-1, the relation nodes attached to the Solvent's Mass are also instantiated. As a result, the instantiated relation node ADD/SUB is further instantiated, i.e., the arg-2 is bound to 98 gr. Since both arg-1 and arg-2 are known in this instantiated relation node, one of the procedures defined in this relation node which computes arg-3 becomes to be evaluable. The procedure is marked as

evaluable and put into a list called EV-List (See 4.2). In the sequel, the final result will be such as shown in Fig. 12.

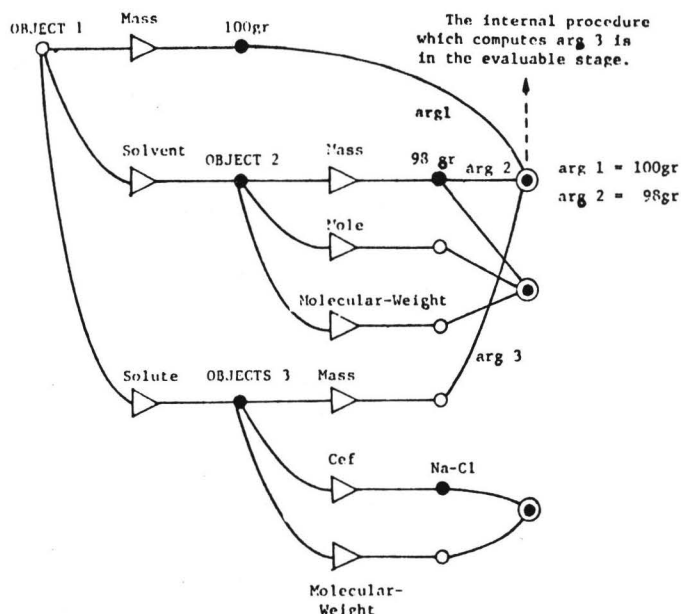


Fig. 12. Final Result of the Instantiation.

As described above, the instantiation of a data type recursively causes the instantiation of other data types. Some relation nodes are also instantiated during the process. The instantiation of relation nodes is propagated upward through the SORT-hierarchy: If a property of a certain data type is filled, the relation nodes attached to the same property of the higher data types in the hierarchy are all instantiated.

4.2 Backward and Forward Reasonings

As we described earlier, after the initial instantiation of variables, a certain set of procedures are only marked as immediately evaluable and put into the EV-list, because the immediate evaluation of these procedures will trigger the unlimited forward reasoning which may invoke many irrelevant procedures.

The next stage is performed backward. The initial goal, which is to solve the problem posed by the user, will be tried, and if not fulfilled, the goal will be further expanded into a set of subgoals. The expansion of a goal will be performed by utilizing the information attached to the instantiated data types, especially the declaration parts of internal procedures in the instantiated relation nodes. Because the instantiated relation nodes have already been

partially instantiated during the preceding forward mode, it is reasonable to expect that these instantiated relationships are relevant to the current problem more than those which have not been instantiated. We illustrate this process by considering a simple example. Suppose the problem description in 4.1 has been given and that the following problem is posed.

'Retrieve the Density of OBJECT-1'

The initial goal will not be achieved, because the mass of OBJECT-3 is not explicitly given. The initial goal is expanded into subgoals. Since the relation node which indicates the relationship among Solution's Mass, Solute's Mass and Solution's Density has been partially instantiated and attached to the instance OBJECT-1, this instantiated relation node will be used to expand the goal. The subgoal is

'Retrieve the Mass of OBJECT-3'.

The reasoning mode will be changed from the backward mode to the forward mode. In this mode, the procedures in the EV-list will be evaluated in turn. The evaluation will further instantiate some variables in the S-Net so that some other procedures will enter in the evaluable stage. They are also marked accordingly, and put in the EV-list. They will be evaluated in the next forward mode.

In the above example, the internal procedure in the relation node ADD/SUB, which computes the Solute's(OBJECT-3's) Mass, is evaluated. The result of the evaluation fulfills the above subgoal. This will be checked in the next backward mode. The success of the subgoal will result in the success of the initial goal.

Thus, the whole process is performed forwards and backwards. If there are no procedures in the EV-list, only the backward mode will be continued until a certain procedure will be entered in the EV-list. Note that an internal procedure in a relation node will be used in both directions, forwards and backwards. Moreover, the two mode works cooperatively in the sense that the backward mode prefers to operating the partially instantiated relation nodes proposed by the preceding forward mode.

5. A SIMPLE EXAMPLE

In order to give the reader some idea about the operation of the problem solver, we will give an example in this section. Suppose the following problem is given.

```
(OBJECT-1      NAME = Solution
  (!  Solute) = ( OBJECT-2)
  (!  Volume) = 200 cc )
(OBJECT-2      NAME = Sodium
  (!  Mass) = 2 gr )
(RETRIEVE      OBJECT-1  Mol-Density) ?
```

In order to compute the Mol-Density, the problem solver should solve various subproblems such as computing the Solute's Mol-Number from the mass and the molecular-weight, retrieving the Molecular-Weight from the external data base, getting Sodium's Chemical-Formula for accessing the data base etc. The problem solver works fairly well and selects relevant knowledge appropriately even if there is much irrelevant knowledge in the S-Net.

6. CONCLUSION

In this paper, we have proposed a graphical notation called S-Net which provides a framework and a foundation for some of the operations of recent AI languages and systems. Especially, it is shown that the problems, 'when an attached procedure to be invoked' and 'how to represent restricted property inheritance in nonprocedural forms', can be solved in the S-Net representation. Moreover, various sorts of information about problem solving situations such as variable binding contexts etc. are distributively stored in the nodes(partially instantiated relation nodes etc.) so that the global controller, the problem solver, can be easily implemented. The current version is only one of possible constructions. We are now developing different problem solving strategies.

REFERENCES

- [Bobrow 1976] D.G.Bobrow et.al., "An Overview of KRL, a Knowledge Representation Language," Technical Report, Xerox PARC, 1976
- [Davis 1978] R.Davis, "Knowledge Acquisition in Rule-Based Systems," in Pattern Directed Inference Systems (eds. Waterman & Hayes-Roth), Academic Press, 1978
- [Goldstein 1977] I.P.Goldstein et.al., "The FRL Primer," AI Memo 408, MIT, 1977
- [Hendrix 1975] G.G.Hendrix, "Partitioned Network for the Mathematical Modeling of Natural Language Semantics," Ph.D Thesis, Univ. of Texas, 1975
- [Minsky 1975] M.Minsky, "A Framework for Representing Knowledge," in the Psychology of Computer Vision (ed. Winston), McGraw Hill, 1975
- [Schubert 1976] L.K.Schubert, "Extending the Expressive Power of Semantic Networks," Artificial Intelligence, Vol. 7, 1976