1981-14

SNePSLOG
A "Higher Order" Logic Programming Language

Stuart C. Shapiro, Donald P. McKay
Joao Martins & Ernesto Morgado

SNeRG
Technical Note No. 8
Department of Computer Science
State University of New York at Buffalo
Amherst, NY, 14226
August, 1981

This work was supported in part by the National Science Foundation under Grant No. MCS80-06314.

1. INTRODUCTION

SNePSLOG [2] is a logic programming interface to SNePS, the Semantic Network Processing System [6], and SNIP, the SNePS Inference Package. Assertions and rules written in SNePSLOG are stored as structures in a semantic network. SNePSLOG queries are translated into top-down deduction requests to SNIP. Assertions can also be stated in a way that triggers bottom-up SNIP deductions. Output from SNIP is translated into SNePSLOG formulas for printing to the user.

Since SNePSLOG predicates and formulas are represented as nodes in the semantic network, and since SNIP allows variables to range over any nodes, SNePSLOG expressions are not limited to first order predicate calculus. Examples in this paper will show SNePSLOG rules that quantify over predicates, and a SNePSLOG rule that has a rule in antecedent position which is treated like an

atomic assertion during deduction.

Since SNIP supports several non-standard logical constants [6; 8; 9], the SNePSLOG syntax also allows them. Some of these are used below and will be explained where they first occur.

The examples in this paper refer to naval information and were taken from the text and data of [5].

2. EXAMPLES OF SNePSLOG

2.1. Representing Set Partitions and Concept Names

In a taxonomy, a set is partitioned into several disjoint subsets, each one containing elements which share similar properties. For example, the Long Beach class of ships have the same weapon and sensor suites. The use of disjoint subsets increases the power of the taxonomy by enabling the system to prove negative assertions. For example, the system can prove that the Long Beach is not a destroyer since it knows that the Long Beach is a cruiser, and, since cruisers and destroyers are in the partitioning of ships according to ship-type, they are disjoint sets.

A single set can be categorized by several different partitionings. Ships are categorized by ship-type (cruiser, destroyer, etc.), as well as by ship-class (Long Beach Class, California Class, Belknap Class, etc.). Every ship is in exactly one ship-type and exactly one ship-class. We use the assertion

partitioning-of(p,s) to mean that the set p is a partitioning of the set s, the assertion p(r) to mean that the set r is a member of the partitioning p, and the assertion r(x) to mean that x is a member of the set r. Notice that these assertions are not first order, and that they imply that x is a member of s.

In the examples below, lines beginning with the prompt ">" are user input and ";" marks the beginning of a comment.

```
> ; partitioning conveyances
> ; ------------------------
> ;
> partitioning-of(conveyance-type@,conveyance@)
> ;
> conveyance-type@(ship@)
> conveyance-type@(sub@)
> conveyance-type@(plane@)
> conveyance-type@(helo@)

> ; partitioning ships into types
> ; -----------------------------
> ;
> partitioning-of(ship-type@,ship@)
> ship-type@(cruiser@)
> ship-type@(destroyer@)

> ; partitioning ships into classes
> ; -------------------------------
> ;
> partitioning-of(ship-class@,ship@)
> ship-class@(long-beach-class@)
> ship-class@(california-class@)
> ship-class@(belknap-class@)

> ; partitioning cruisers
> ; ---------------------
> ;
> partitioning-of(cruiser-type@,cruiser@)
> cruiser-type@(nuclear-cruiser@)
> cruiser-type@(fossil-fuel-cruiser@)

> ; partitioning nuclear cruisers
> ; -----------------------------
> ;
> partitioning-of(nuclear-cruiser-class@,nuclear-cruiser@)
> nuclear-cruiser-class@(long-beach-class@)
> nuclear-cruiser-class@(california-class@)

> ; partitioning fossil-fuel cruisers
> ; ---------------------------------
> ;
```

```
> partitioning-of(fossil-fuel-cruiser-class@,
>                 fossil-fuel-cruiser@)
> fossil-fuel-cruiser-class@(belknap-class@)

> ; ships
> ; -----
> ;
> long-beach-class@(long-beach@)
> california-class@(california@)
> california-class@(south-carolina@)
> belknap-class@(belknap@)
```

When considering ships and ship classes, it is necessary to distinguish the name of an object and the object itself because different objects can share the same name. This feature allows a user to ask if the California is a California, where the first name refers to the ship named California and the second to the ship-class named California. For example, the rule

```
> ALL(x,xname,y,yname)
>    [name-of(xname,x), name-of(yname,y)]
>       &=> {[y(x) V=> is-a(xname,yname)]
>            [~y(x) V=> ~is-a(xname,yname)]}
```

asserts that the "is-a" relation holds, or not, depending on the underlying relationship of the named entities. Every entity which has a suffix "@" and needs to be referenced by a name has the appropriate name-of assertion, e.g. name-of(California, california@) and name-of(California, california-class@). These are used as follows. (Lines following a query are abbreviated system responses.)

```
> ?is-a(California, California)
 SINCE
 california-class@(california@) name-of(California,california@)
 name-of(California,california-class@)
 WE INFER
 is-a(California,California)
```

The above rule demonstrates SNePSLOG's ability to handle second order rules, and uses the connectives ~, &=>, and V=>. ~

is the standard negation. $A_1, \ldots, A_n$ &=> $C_1, \ldots, C_m$ asserts that the conjunction of $A_1, \ldots, A_n$ implies the disjunction of $C_1, \ldots, C_m$. $A_1, \ldots, A_n$ V=> $C_1, \ldots, C_m$ asserts that the disjunction of $A_1, \ldots, A_n$ implies the disjunction of $C_1, \ldots, C_m$.

As an adequate representation for reasoning with partitions, we use two rules. The first expresses the normal set inclusion property of partitions:

```
> ALL(x,p,p-ing,s)
>      [partitioning-of(p-ing,s), p-ing(p)]
>           &=> [p(x) V=> s(x)]
```

which asserts that if p-ing is a partitioning of a set s and p is an element of p-ing then if x is an element of p then it's an element of s as well. For example, this rule allows the proposition that the California is a conveyance to be deduced because conveyance type is a partitioning of conveyances, ship is a conveyance type and the California is a ship. (The "1" after the "?" tells the system to suspend deduction as soon as one answer is found.)

```
 > ?1 is-a(California, conveyance)
SINCE
name-of(California,california@) name-of(conveyance,conveyance@)
WE INFER
(conveyance@(california@) V=> is-a(California,conveyance))

SINCE
name-of(California,california-class@)
name-of(conveyance,conveyance@)
WE INFER
(conveyance@(california-class@) V=> is-a(California,conveyance))


SINCE
conveyance-type@(ship@)
partitioning-of(conveyance-type@,conveyance@)
WE INFER
(ship@(california@) V=> conveyance@(california@))

SINCE
```

```
ship@(california@)
WE INFER
conveyance@(california@)

SINCE
cõnveyance@(california@)
WE INFER
is-a(California,conveyance)
```

A second rule expresses the disjointness property:

```
> ALL(x,s,p-ing)
>       [partitioning-of(p-ing,s), s(x)]
>          &=> NEXISTS(_,1,_)(p)[p-ing(p) : p(x)]
```

which states that if p-ing is a partitioning of a set s and x is an element of s then there is at most one p such that p is an element of p-ing and x is in p. NEXISTS(i,j,n)(x)[A_1(x), ..., A_m(x): C(x)] uses the numerical quantifier [8] and asserts that of the n individuals x which satisfy A_1(x), ..., A_m(x), at least i and at most j also satisfy C(x). In this case, only the j parameter is being used. That the Long Beach is not a destroyer can be deduced using this rule because ship-type is a partitioning of ships, the Long Beach is a ship, both cruiser and destroyer are ship-types and the Long Beach is known to be a cruiser.

```
> ?1 is-a(Long-Beach,destroyer)
SINCE
partitioning-of(nuclear-cruiser-class@,nuclear-cruiser@)
nuclear-cruiser-class@(long-beach-class@)
long-beach-class@(long-beach@)
WE INFER
nuclear-cruiser@(long-beach@)

SINCE
partitioning-of(ship-class@,ship@)
ship-class@(long-beach-class@) long-beach-class@(long-beach@)
WE INFER
ship@(long-beach@)

SINCE
nuclear-cruiser@(long-beach@)
partitioning-of(cruiser-type@,cruiser@)
cruiser-type@(nuclear-cruiser@)
```

```
WE INFER
cruiser@(long-beach@)

SINCE
ship@(long-beach@) partitioning-of(ship-type@,ship@)
WE INFER
NEXISTS((_,1,_),(p))(ship-type@(p) : p(long-beach@))

SINCE
ship-type@(destroyer@)
WE INFER
~destroyer@(long-beach@)
```

One advantage of explicity representing partitions is that the system is able to answer questions about how the hierarchy is arranged. Thus, the system is able to answer questions such as "What are the partitionings of ships ?" and answer with "The set of ships is partitioned into ship classes and ship types." This is demonstrated with the following query. (The character "%" flags a free variable in a query.)

```
> ? partitioning-of(%p, ship@)

WE KNOW
partitioning-of(ship-class@,ship@)
WE KNOW
partitioning-of(ship-type@,ship@)
```

2.2. Bi-directional Inference and Default Reasoning

Bi-directional inference [1] occurs when the user does top-down inference followed by bottom-up inference. It is characterized by the focusing of the deductions towards previously asked questions. In other words, bottom-up inference does not result in the deduction of everything that could be deduced but rather in the deduction of everything which is

relevant to previously asked questions.

We first enter information about the weapons and sensors of the Converted Forrest Sherman (CFS) class.

```
>  ; Ships in the Converted Forrest Sherman Class have weapons
>  ; of type CFS-W and sensors of type CFS-S.
>  ;
>  ALL(S)[ Class(S,Converted-Forrest-Sherman)
>          V=>
>          (Weapons(S,CFS-W), Sensors(S,CFS-S))]

>  ; The weapons of type CFS-W are one single Tartar, one
>  ; Mk-42, one ASROC 8-tube and two Mk-32.
>  ;
>  ALL(S)[ Weapons(S,CFS-W)
>          V=>
>          ( Has(S,one,Single-Tartar),
>            Has(S,one,Mk-42),
>            Has(S,one,ASROC-8-tube),
>            Has(S,two,Mk-32) )]

>  ; Single Tartar and ASROC 8-tube are launchers,
>  ; Mk-42 is a gun, and Mk-32 are torpedo tubes.
>  ;
>  Launcher(Single-Tartar)
>  Launcher(ASROC-8-tube)
>  Gun(Mk-42)
>  Torpedo-tubes(Mk-32)

>  ; The sensors of type CFS-S are one SQS-23, one SPS-10,
>  ; either one of SPS-37 or SPS-40 and one SPS-48.
>  ;
>  ALL(S)[ Sensors(S,CFS-S)
>          V=>
>          ( Has(S,one,SQS-23),
>            Has(S,one,SPS-10),
>            ANDOR(1,1)( Has(S,one,SPS-37),
>                        Has(S,one,SPS-40) ),
>            Has(S,one,SPS-48) )]

>  ; By default, the sensors of type CFS-S have one SPS-37
>  ;
>  ALL(S)[ Sensors(S,CFS-S) V=>  DELTA(Has(S,one,SPS-37)) ]

>  ; SPS-10, SPS-37, SPS-40 and SPS-48 are radars
>  ; and SQS-23 is a sonar
>  ;
>  Radar(SPS-10)
>  Radar(SPS-37)
>  Radar(SPS-40)
```

```
> Radar(SPS-48)
> Sonar(SQS-23)
```

In these rules, we have seen two new connectives. ANDOR(i,j)(A_1, ..., A_n) asserts that at least i and at most j of A_1, ..., A_n are true, so ANDOR(1,1) is exclusive or. DELTA is a default operator. DELTA(A) asserts that if ~A is not derivable in the current data base, A should be deduced. This particular default rule is used because the text says that all ships in the CFS class have one SPS-37 except the Somers, which has one SPS-40 instead.

To see bottom-up reasoning, we assert that the John Paul Jones is in the CFS class using the "!" command. (The tracing is not shown here.)

```
> ; John-Paul-Jones is of class
> ; Converted-Forrest-Sherman!
> ;
> Class(John-Paul-Jones,Converted-Forrest-Sherman) !

Surface description of value:
Class(John-Paul-Jones,Converted-Forrest-Sherman)
Weapons(John-Paul-Jones,CFS-W)
Sensors(John-Paul-Jones,CFS-S)
Has(John-Paul-Jones,one,Single-Tartar)
Has(John-Paul-Jones,one,Mk-42)
Has(John-Paul-Jones,one,ASROC-8-tube)
Has(John-Paul-Jones,two,Mk-32)
Has(John-Paul-Jones,one,SQS-23)
Has(John-Paul-Jones,one,SPS-10)
Has(John-Paul-Jones,one,SPS-48)
Has(John-Paul-Jones,one,SPS-37)
~Has(John-Paul-Jones,one,SPS-40)
```

Notice that the defult rule and the exclusive or rule were used.

To deonstrate bi-directional inference, we first ask if the Decatur has an SPS-10. The system can't answer because it never

heard of the Decatur before. We then assert that the Decatur is in the CFS class using the "!" command. Above, making this assertion about the John Paul Jones resulted in eleven inferred facts. Here, because of the influence of the previous query, only two facts are inferred.

```
> ; Does Decatur have one SPS-10?
> ;
> Has(Decatur,one,SPS-10) ?

   Surface description of value:
> ; No answer given, indicating "I don't know"

> ; Decatur is of class Converted-Forrest-Sherman!
> ;
> Class(Decatur,Converted-Forrest-Sherman) !
>

SINCE
Class(Decatur,Converted-Forrest-Sherman)
WE INFER
Sensors(Decatur,CFS-S)


SINCE
Sensors(Decatur,CFS-S)
WE INFER
Has(Decatur,one,SPS-10)

   Surface description of value:
   Class(Decatur,Converted-Forrest-Sherman)
   Sensors(Decatur,CFS-S)
   Has(Decatur,one,SPS-10)
```

Above, we saw the default rule used to infer that the John Paul Jones has one SPS-37 and no SPS-40. We now tell the system about the exceptional Somers, and ask the system if it, nevertheless, believes that the Somers has one SPS-37.

```
> ; Somers is in the Converted Forrest Sherman class
> ; but has one SPS-40.
> ;
> Class(Somers,Converted-Forrest-Sherman)
> Has(Somers,one,SPS-40)

> ; Does Somers have one SPS-37 radar?
> ;
```

```
> Has(Somers,one,SPS-37) & Radar(SPS-37) ?

SINCE
Class(Somers,Converted-Forrest-Sherman)
WE INFER
Sensors(Somers,CFS-S)


SINCE
Sensors(Somers,CFS-S)
WE INFER
DELTA(Has(Somers,one,SPS-37))


SINCE
Sensors(Somers,CFS-S)
WE INFER
ANDOR(1,1)(Has(Somers,one,SPS-37),Has(Somers,one,SPS-40))

SINCE
Has(Somers,one,SPS-40)
WE INFER
~Has(Somers,one,SPS-37)

Surface description of value:
Sensors(Somers,CFS-S)
~Has(Somers,one,SPS-37)
Has(Somers,one,SPS-40)
```

2.3. Using rules as data

This example shows that rules can be treated as data. The information in this example is mainly from the following paragraph of [5, p. 7]:

"PLANES are like ships, except that they have an ALTITUDE, a PILOT, a SQUADRON, and a HOME-SHIP. PLANES do not have DRAFT, DISPLACEMENT or NAME. PLANES also have types and classes, and have TAIL-NUMBERS in place of HULL-NUMBERS."

The main point to note about this example is that in the rule about "Like" there is a rule in antecedent position which is

treated as data during the deduction process: when the system is asked whether the Flying Dutchman has a cruise speed it tries to find out whether all ships have a cruise speed and since this rule is explicitly asserted in the network it is used as data. Also note that this rule is expressed as a second order rule -- X1, X2, and R are quantified variables used as predicates.

```
>  ; Ships have a Cruise speed and a Draft.
>  ;
>  ALL(S)[Ship(S) V=> (Has-a(S,Cruise-speed),Has-a(S,Draft))]

>  ; We define the relation "Like" by saying that
>  ; if X1 is like X2, then if all the elements of X2 have the
>  ; relation R to some P, then, by default
>  ; so have the elements of X1.
>  ;
>  ALL(X1,X2)[ Like(X1,X2)
>              V=>
>              ALL(R,P)[ (ALL(E2)[ X2(E2) V=>  R(E2,P) ])
>                        V=>
>                        ALL(E1)[ X1(E1) V=>  DELTA(R(E1,P)) ] ]]

>  ; Planes are like ships ...
>  ;
>  Like(Plane,Ship)

>  ; .. except that they have an altitude, a pilot, a squadron
>  ; and a home-ship.
```

This rule uses the THRESH connective. THRESH(i)(A_1, ..., A_n)

asserts that either fewer than i or all n of

A_1, ..., A_n are true. THRESH(1) says that all the

arguments have the same truth value,

so this rule states that having an altitude, a

pilot a squadron and a home-ship are characteristics of

planes in the sense that if we know that an object has

one of these properties we can conclude that it is a

plane and that it has all the others.

```
>  ALL(P) [ THRESH(1) ( Plane(P), Has-a(P,Altitude),
>                       Has-a(P,Pilot), Has-a(P,Squadron),
>                       Has-a(P,Home-ship) )]

>  ; Planes do not have draft, displacement or name.
>  ;
>  ALL(P) [ Plane(P) V=>  ANDOR(0,0) ( Has-a(P,Draft),
>                                      Has-a(P,Displacement),
>                                      Has-a(P,Name) )]
>  ; This use of ANDOR is as an extended nor.

>  ; Tail numbers and Hull numbers are mutually exclusive
>  ;
>  ALL(X) [ ANDOR(0,1)(Has-a(X,Tail-number),
>                      Has-a(X,Hull-number))]
>  ; This use of ANDOR is as a nand.

>  ; Planes have tail numbers
>  ;
>  ALL(P) [ Plane(P) V=>  Has-a(P,Tail-number) ]

>  ; Flying Dutchman is a plane
>  ;
>  Plane(Flying-Dutchman)

>  ; Does Flying-Dutchman have a cruise-speed?
>  ;
>  Has-a(Flying-Dutchman,Cruise-speed) ?

SINCE
Like(Plane,Ship)
WE INFER
((Ship(E2) V=>  Has-a(E2,Cruise-speed))
V=>
(Plane(Flying-Dutchman)
 V=>
 DELTA(Has-a(Flying-Dutchman,Cruise-speed)))))


SINCE
(Ship(S) V=>  Has-a(S,Cruise-speed))
WE INFER
(Plane(Flying-Dutchman)
 V=>
 (DELTA(Has-a(Flying-Dutchman,Cruise-speed))))
```

```
SINCE
Plane(Flying-Dutchman)
WE INFER
Has-a(Flying-Dutchman,Cruise-speed)

>  ; Does Flying-Dutchman have a Hull Number?
>  ;
>  Has-a(Flying-Dutchman,Hull-number) ?

SINCE
Like(Plane,Ship)
WE INFER
((Ship(E2) V=>  Has-a(E2,Hull-number))
V=>
(Plane(Flying-Dutchman)
 V=>
 DELTA(Has-a(Flying-Dutchman,Hull-Number))))


SINCE
Plane(Flying-Dutchman)
WE INFER
Has-a(Flying-Dutchman,Tail-number)


SINCE
Has-a(Flying-Dutchman,Tail-number)
WE INFER
~Has-a(Flying-Dutchman,Hull-number)


Surface description of value:
Has-a(Flying-Dutchman,Tail-number)
~Has-a(Flying-Dutchman,Hull-number)

>  ; Does Flying-Dutchman have a Draft ?
>  ;
>  Has-a(Flying-Dutchman,Draft) ?

SINCE
Like(Plane,Ship)
WE INFER
((Ship(E2) V=>  Has-a(E2,Draft))
V=>
(Plane(Flying-Dutchman)
 V=>
 DELTA(Has-a(Flying-Dutchman,Draft))))


SINCE
Plane(Flying-Dutchman)
WE INFER
~Has-a(Flying-Dutchman,Draft)

Surface description of value:
```

~Has-a(Flying-Dutchman,Draft)

3. IMPLEMENTATION

SNePS and its supporting systems are currently implemented in ALISP, a dialect of LISP, on the CYBER 174 at SUNY/Buffalo. A version of the system was translated into INTERLISP by Bob Bechtel, of the Naval Ocean Systems Center, San Diego, California. An abstract view of SNIP is in [4]. The simulated multiprocessing system which supports it is described in [3]. SNePSLOG is currently implemented as an Augmented Transition Network parser-generator grammar [7] which acts as a user interface with the rest of the system.

4. SUMMARY

SNePSLOG is a logic programming language implemented as a user interface to SNePS, the Semantic Network Processing System, and SNIP, the SNePS Inference Package. Examples of SNePSLOG were shown featuring higher-order rules, a rule that treats rules as data (a meta-rule), bottom-up, top-down, and focussed bi-directional inference, and non-standard connectives. These features derive from the semantic network representation used for assertions and rules.

REFERENCES

1. Martins J., McKay D. and Shapiro S., Bi-Directional Inference, Technical Report 174, Dept. of Computer Science, State University of New York at Buffalo, 1981.

2. McKay D. and Martins J., Provisional SNePSLOG User's Manual, Dept. of Computer Science, State University of New York at Buffalo, 1981.

3. McKay D. and Shapiro S., MULTI - A LISP Based Multiprocessing System, Proc. 1980 LISP Conference, 29-37.

4. McKay D. and Shapiro S., Using Active Connection Graphs for Reasoning with Recursive Rules, Proc. Seventh International Joint Conference of Artificial Intelligence, 1981.

5. Naval Ocean Systems Center, Elements of Naval Domain Knowledge, NOSC Working Paper, San Diego, CA, April, 1981.

6. Shapiro, S.C., The SNePS semantic network processing system. In Associative Networks: The Representation and Use of Knowledge by Computers, Findler, N.V., ed., Academic Press, New York, 1979, 179-203.

7. Shapiro, S.C., Generalized augmented transition network grammars for generation from semantic networks, Proc. 17th Annual Meeting of the Association for Computational Linguistics, University of California at San Diego, August, 1979, 25-29.

8. Shapiro, S.C., Numerical quantifiers and their use in reasoning with negative information. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Computer Science Dept., Stanford University, Stanford, CA, 1979, 791-796.

9. Shapiro S., Using Non-Standard Connectives and Quantifiers for representing Deduction Rules in a Semantic Network, presented at "Curre Aspects of AI Research" a seminar held at the Electrotechnical Laboratory, Tokyo, August 22-27, 1979.