INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Development of Techniques to Control Reasoning using SNePS

Pedro Miguel Lopes das Neves

Dissertação para a obtenção do Grau de Mestre em

Engenharia Informática e Computadores

Júri

Presidente: Professor Doutor Ernesto José Marques Morgado

Orientador: Professor Doutor João Emílio Segurado Pavão Martins

Vogais: Professora Doutora Maria dos Remédios Vaz Pereira Lopes Cravo

Professor Doutor Nuno João Neves Mamede

Novembro de 2007

Acknowledgments

I would like to thank to Professor João Pavão Martins, who spent several hours guiding me throughout this work. I am very grateful for his enormous patience and for his contagious investigating spirit.

I also want to thank to my parents for all their support and kindness. Thank you for the so many big opportunities you gave me and for constantly encouraging me during my life as a student.

Resumo

O SNePS (Semantic Network Processing System)[1] é um formalismo de representação de conhecimento que implementa um mecanismo de inferência baseada em nós. Nesta rede associam-se processos a nós que comunicam através de canais, executando regras de inferência não-standard e permitindo acreditar novos nós.

Apesar de ter propriedades interessantes, o mecanismo de inferência do SNePS (SNIP) não é o principal ramo de desenvolvimento dos investigadores deste sistema, mais preocupados com a representação de expressões em língua natural.

Outros sistemas automáticos de dedução tais como a Resolução, nasceram com a preocupação de encontrar demonstrações para teoremas. A Resolução sofreu melhoramentos ao longo do tempo, fazendo dela mais poderosa e eficiente. O PROLOG, um sistema de programação em lógica, procura refutações lineares usando uma função de selecção e assumindo cláusulas determinadas.

Neste trabalho desenvolvemos mecanismos para controlar a inferência feita pelo SNePS. Para isso descrevemos e comparamos o SNIP com o PROLOG, com a intenção de encontrar as vantagens e desvantagens de cada mecanismo. A expressividade dos dois sistemas, as suas regras de inferência, a quantidade de deduções redundantes efectuadas, as estratégias de pesquisa usadas e a forma como processam regras recursivas são pontos de comparação explorados.

Esta comparação sugere alguns melhoramentos ao SNIP, que são implementados e testados. Um mecanismo para suspender a execução de alguns processos sem afectar o número de soluções encontradas é proposto e a sua fundamentação teórica apresentada. Adicionalmente uma forma de parametrizar a estratégia de pesquisa é sugerida, tornando o processo de inferência mais versátil.

Palavras Chave: rede semântica, SNePS, resolução, PROLOG, inferência regressiva, dedução automática.

[1] Em Português, Sistema de Processamento de Redes Semânticas.

Abstract

SNePS, Semantic Network Processing System, is a knowledge representation formalism that implements a node-based inference mechanism. Processes are associated to nodes, communicate through channels, and execute non-standard inference rules, allowing the assertion of new nodes.

Despite having interesting properties (avoidance of redundant deductions, non circularity on recursive rules, potential of parallelization of deduction, among others), SNePS inference capabilities are not the main trend of research of SNePS developers. They are more focused on the representational capabilities of natural language expressions using this formalism.

Other automatic deduction systems such as Resolution, were born with the primary intention of finding proofs for theorems. Resolution went through many improvements across time, making it more powerful and efficient. PROLOG, a logic programming system, searches for linear refutations using a selection function and assuming only definite clauses.

On this work we develop mechanisms to control the inference made by SNePS. For that we describe and compare the SNePS Inference Package (SNIP) with PROLOG, with the intention of finding advantages and drawbacks of both inference mechanisms. The expressiveness of both systems, their inference rules, the amount of redundant deductions made, the search strategy used by each of them and the way they handle recursive rules are points of comparison explored.

This comparison suggests SNIP improvements that are implemented and tested. A method to idle some processes, without affecting the number of solutions found, is proposed and its theoretical foundation explained. Additionally a way to parameterize the search strategy is suggested, making inference process more versatile.

Keywords: semantic network, SNePS, resolution, PROLOG, backward inference, automatic deduction.

Table of Contents

List of Tables

List of Figures

1. Introduction

The Semantic Network Processing System (SNePS) (Shapiro, 1979) is a knowledge representation formalism that uses a propositional and assertional semantic network with the main intention of processing natural language and performing commonsense reasoning (Shapiro, 1999). It is a system that has been under development during the last decades and has passed through several improvements (Shapiro, et al., 1992).

Besides the network-based representation, SNePS has several components, namely:

- SNIP, the SNePS Inference Package, is capable of applying path-based and node-based inference to assert new nodes that are implied by the network (Hull, 1986);
- SNePSLOG, a logical interface to SNePS, receives higher order logical formulas, parses those formulas and represent them on the semantic network using predefined case-frames (Shapiro, et al., 1981);
- SNeBR, which is a belief revision system, allows reasoning on multiple belief spaces (Martins, et al., 1983);
- SNaLPS, the SNePS Natural Language Processing System, can establish a conversation with the user in English and answer questions when asked (Shapiro, et al., 1982).

From all these components we will focus mainly on the first one[2], SNIP, and from all the inference methods available we will explore node-based inference. This SNePS' inference mechanism attaches a process to each node on the network. Each node has the ability to match other nodes and can interact with them exchanging messages through channels. The processes share new solutions under a producers-consumers model and are scheduled to work under a given policy.

Although expressiveness is a key feature of SNePS, since the system is concerned with the easiness of representing knowledge, the automatic deduction system implemented by SNIP has several interesting features. SNIP can handle recursive rules in an interesting manner, avoids redundant deductions, allows, to a great extent, parallelization of deduction and produces a human-like reasoning structure. However it lacks efficiency, ability to control the inference process and versatility on the search strategy used.

Other inference systems had their origin on the pure intention of automatically proving theorems. These systems are more concerned with properties such as completeness and efficiency, than with the easiness of representation. Resolution is one of those cases and is not only one of the oldest automatic deduction systems but also one of the most commonly used nowadays on state-of-art provers. The initial method went through an interesting story of continuous improvements making it more powerful. PROLOG, a logic programming system, is based on SLD-resolution, a restriction of the original method to linear deductions, with Horn clauses and a selection function.

[2] SNePSLOG will also help us to more easily represent first-order formulas on the network.

The analysis of these two inference systems allows us to conclude an interesting tradeoff between expressiveness and complexity of the inference rules: more expressiveness implies more complex inference rules. Additionally, more complexity on inference rules implies a more complex deduction process, and a more complex deduction process generally means it is less efficiency and less controllable.

Our work addresses this dichotomy, comparing SNIP and PROLOG with two main goals in mind: on the one hand, the comparison guides our approach to SNePS in the direction we are willing to, i.e., we want to emphasize the automatic deduction capabilities of SNePS; on the other it provides hints and suggestions for possible improvements to SNIP.

After this theoretical effort, we implemented some ideas on SNePS and test their practical use. Two improvements are suggested: we try to avoid the execution of some processes without affecting the number of possible derivable solutions and we show how the search strategy used by SNIP can be parameterized by the user.

Given all this, it is now possible to formulate the objective of this work as *the development of techniques to control the inference process of SNePS and its potential improvement*. The adjective potential is important here, as we are not looking only for great improvements but rather for a study of this system.

This document is arranged on five chapters. We describe the inference process of SNePS; then we shift our attention to the well known logic programming system, PROLOG, starting from ground resolution and reaching SLD-resolution through a series of resolution refinements; we present an explicit comparison between SNIP and PROLOG, focusing on aspects that can be subjected to changes; finally, on the last two chapters, we suggest some improvements to SNIP, carefully describing the problems we want to tackle, explaining the theoretical foundations of the solutions, enumerating the changes made to the SNePS' code and giving some results.

2. Understanding SNePS

The definition of semantic network is not consensual. On this work, although we do not want to provide an universal definition of the term, the fact that "virtually every networklike formalism that appeared in the literature since 1966 has at one time been branded by someone a semantic net" (Brachman, 1979) lead us to first of all, try to understand the potential and limitations of SNePS when compared with other formalisms of this type. Several comprehensive works like (Sowa, 1991) and (Lehmann, 1992) provide a good overview of this area.

Therefore on this section, and before going deeper in the study of the internal architecture of SNePS, we try to characterize this formalism according to different taxonomies of semantic networks found on the literature. Afterwards we briefly consider the syntax and semantic of the network and in the end we describe the inference mechanism of SNePS.

2.1. So many semantic networks... Where does SNePS fit?

The difficulty of delimiting the extent of the concept "semantic network" arises from its own history. The fact that many different areas of science with obviously different objectives suddenly noticed the potential of these networks, made the term become vague.

Psychologists tried to use semantic networks to understand the way our mind works, to study memory and conjecture the way we structure concepts on our brain. They tried to get experimental evidences, through the measurement of reaction time, of how far on our semantic memory concepts were (Collins, et al., 1969).

Linguists used them to represent sentences and to show the role of each of their components on that sentence. Verbs were the central node around which nouns and agents were placed (Tesnière, 1959).

Mathematicians were interested on the potential of these nets to represent logical formulas (Peirce, 1980).

Finally, computer scientists used them as a formal system that not only allows the representation of knowledge on a computer, but also the development of automated deduction mechanisms over that knowledge. This is the approach we are interested in.

Among so many disparate motivations, the only characteristic that seems to remain constant over all these uses is the existence of a graphical notation of the network. By now, we assume that s*emantic networks* are a knowledge representation formalism that uses *nodes* to represent concepts and *labeled directed arcs* to represent different types of relations between concepts.

We very briefly allude to the work of both (Brachman, 1979) and (Sowa, 1992). While the first offers a consistent characterization based on knowledge representation concerns and on the "semantic" of the semantic network (the nature of the primitive links available on the formalism), the second gives a very broad overview of the different formalisms mixing both knowledge representation and functionality issues.

Sowa identifies six types of semantic networks: definitional, assertional, implicational, executable, learning and hybrid networks. This last type encompasses all those networks that share characteristics belonging to more than one of the first five classes.

Definitional networks are based on *description logics*. These networks, which can abusively be mapped to the epistemological level proposed on (Brachman, 1979), structure concepts on their internal subpieces and give primitives to build hierarchies of concepts and perform inheritance of properties from types to subtypes. KL-ONE is certainly the best example of this type of networks (Brachman, et al., 1985). In KL-ONE, the attributes of each concept are described using *roles* and *structural descriptions* establish the interactions among the different role fillers.

Assertional networks encompass both *logical* and *conceptual* levels from (Brachman, 1979). These networks are focused on representing assertions about concepts, rather than capturing their internal structure. The different nature of the links on the net distinguishes the two different assertional levels introduced by Brachman.

On networks at the logical level, links associate connectives to propositions and propositions to variables and constants. There exists a mapping between logical formulas and its corresponding network representation. Conceptual Graphs (Sowa, 1978) are a good example of this type of networks.

However, networks at the conceptual level assign a deeper semantic meaning to their links. While on the logical level the structure of the networks is guided by the syntax of the logic, here semantic relations take that role. Schank's conceptual dependencies networks are the best example of these networks (Schank, et al., 1969), where primitive acts like PTRANS (meaning the transfer of a physical object from one place to other) and PROPEL (meaning the application of a physical force to an object) show the type of semantic relations.

Implicational networks establish a net of causal relations between nodes (e.g. if node A is connected to node B, then if A happens B will happen too). Bayesian networks assign probabilities to each implication described on the network (Pearl, 1988), and truth-maintenance systems have a crucial role on belief revision on systems like SNePS, maintaining the dependencies between propositions and assuring that the knowledge base remains in a coherent state (Doyle, 1979).

Executable networks have processes attached to nodes and have the capability of changing the network itself through the execution of those processes (Levesque, et al., 1979).

Finally, we state neural networks as an example of *learning networks* (Rumelhart, et al., 1986). These networks assign a weight to each arc on the graph and use algorithms like *backpropagation* to train the net on a set of instances and obtain a function that tries to get the structure of data.[3]

Now that we have an overview of formalisms with different characteristics, it will be easier to understand the features of the semantic network we will use:

[3] Although Sowa considers neural networks as semantic networks this is not from our point of view, easy to accept. In fact, the poor interpretability of the models obtained using neural networks has been pointed out as a major problem of this method on the machine learning community. The semantic behind a set of weights is certainly arguable…

- According to Sowa's characterization, SNePS is an *assertional* network;
- SNePS is a *propositional semantic network* since logical propositions are represented by nodes. Therefore it is settled at the Brachman's *logical* level;
- It is possible to perform inference on SNePS through SNIP, the SNePS Inference Package (Hull, 1986). This automated deduction system attaches processes to nodes on the network. Therefore SNePS is also an *executable* network.

2.2. Syntax and Semantics of SNePS

Since we are primarily concerned with the study of the way SNePS performs inference, in this section we provide references that the interested reader might use both to understand the basics of SNePS and the way knowledge can be represented using it. Thus (Shapiro, 1979) and (Martins, 2005) are references that share the fact that all their authors have actively worked on the development of SNePS.

However to fully characterize SNePS, we must state four important features not yet mentioned which help us to understand the semantics of the network.

The first one has to do with the nature of the concepts represented. In SNePS concepts are intensional entities, which mean that they are "objects of thought". This is the opposite of extensional entities, which make part of our world and can be touched. Therefore "a mind can have two or more objects of thought that correspond to only one extensional object" and can even "have an object of thought that corresponds to no extensional object" (Shapiro, et al., 1987).

The second important characteristic is the *Uniqueness Principle* which states that whenever two nodes represent the same intensional concept, then they should be the same.

Next, arcs represent non-conceptual relations between nodes, meaning that arcs are structural rather than assertional. Knowledge is represented using case-frames, which are syntactically defined structures of arcs that relate nodes/concepts and have a well defined meaning. Some of these case-frames are already available to the user (namely the ones that are associated to the SNePS inference system we will discuss later), others may be defined by him. The meaning of the network is on the structure and not on the singularity of each link.

On Figure 1, we can find an example of a very simple case-frame that can be adopted by the user to expresses the proposition "*Class2* is a subclass of class *Class1*".

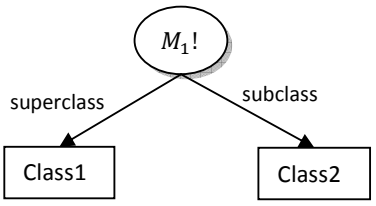


Figure 1 – "Superclass-subclass" case-frame.

From this example, we can start getting acquainted to the notation employing nodes and arcs we will be using throughout this document and understand the non-assertional nature of SNePS' arcs. If instead of the representation of Figure 1, we had defined the arc "is-subclass-of" and used it to connect nodes *Class1* and *Class2*, then this arc would have itself an assertional nature. This is clearly incorrect according to the semantic we adopted for arcs.

Finally, an important consequence of representing propositions using nodes is that we can represent propositions about propositions. Consequently we have a higher order formalism.

2.3. Making implicit knowledge, explicit: the SNePS Inference Package (SNIP)

We approach the task of understanding the way SNePS performs inference, following the strategy:

1. First the different types of inference performed by SNePS are presented. We focus on only one of them.
2. Next we explain how logical connectives can be encoded on our network, which logical connectives are available and their respective inference rules;
3. The next step is to explain the approach used in SNePS to perform unification of nodes;
4. Then we provide a brief explanation of the way deduction processes interact on the network and how they assert new conclusions;
5. Finally we understand MULTI, a LISP based multiprocessing system, to show how the scheduling of these processes is made on SNePS and how inference can be controlled.

2.3.1. Types of inference performed by SNePS

The SNePS' inference package is capable of performing three different types of inference, namely:

- Reduction inference: this method infers, from a given asserted node with *n* arcs, an asserted node with any subset of those arcs (Figure 2).

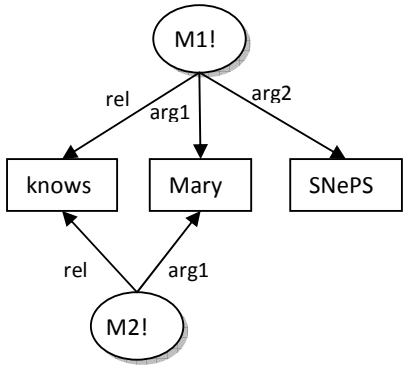


Figure 2 – Reduction inference of "Mary knows" (node M2!) from "Mary knows SNePS".

We can justify this kind of inference facing nodes as a conjunction of binary relations, each expressed by an arc. If this is the case, then we are allowed to infer any subset of that conjunction. On our

example, $M_1!$ is a conjunction of three relations: $rel(M_1!, knows)$, $arg1(M_1!, Mary)$ and $arg2(M_1!, SNePS)$. From this, it is acceptable to infer the conjunction of only the first two, which means to assert node $M_2!$.

- Path-based inference: through this method, we can infer an arc between any two nodes through the specification of a path between them. A path can be defined as a sequence of arcs. Figure 3 shows how the path member-/class/subclass-/class[4] can be used to conclude node M3!.

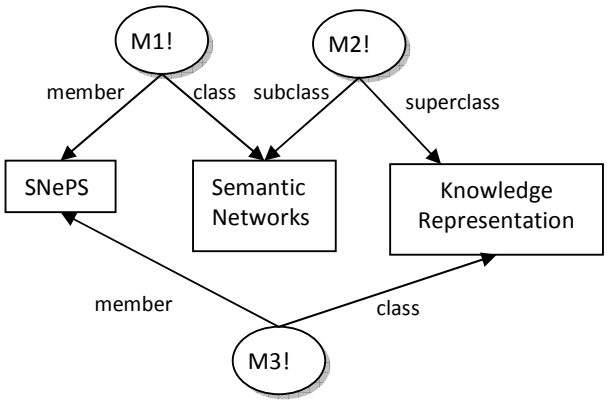


Figure 3 – An inference that "SNePS is a knowledge representation formalism" from "SNePS is a member of class Semantic Network" and "Semantic Networks are a subclass of knowledge representation formalisms".

This kind of inference is strongly connected to inheritance of properties on hierarchy networks(Cravo, et al., 1989). These networks allow to express classes, sub-classes and instances of classes through the specification of different kinds of relations between nodes. A well defined semantics rules the set of possible conclusions that can be inferred from a set of relations.

- Node-based inference: to use this type of inference, a predefined set of rule nodes is available (case frames). Each rule node has a deduction rule associated and connects a set of antecedent nodes to a set of consequents. Whenever an asserted node is matched to an *antecedent node*, then the corresponding node on the *consequent* position can be *forward inferred*. A *backward inference* mechanism is also possible, matching *consequents* and trying to derive antecedents. This method is only possible when an appropriate binding for pattern and variable nodes exists.

A comparison of path-based inference and node-based inference can be found on (Shapiro, 1978).

The rest of this work will focus on **node-based backward inference**.

2.3.2. Rule nodes as non-standard connectives

Although many propositional semantic networks use direct translations of logical connectives like implication, negation and conjunction to nodes on the network - (Schubert, et al., 1979) is a good example of this approach - on SNePS a new set of *non-standard connectives* was devised.

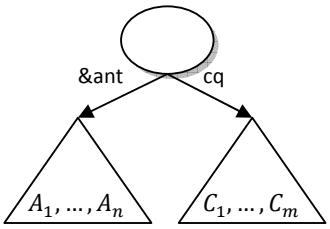
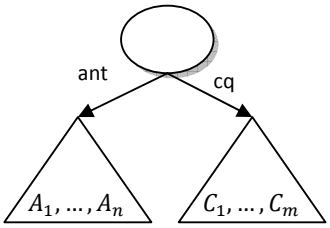
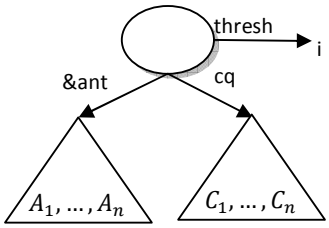
[4] "member-" and "subclass-" denote the converse arcs of "member" and "subclass" respectively.

The motivations behind this approach can be found on (Shapiro, 1979) and are summarized on the list below:

- Easiness of representation: connectives should be expressive and should easily represent natural language expressions on the network.
- Suitability to automation of reasoning: deduction rules should be sound, easily implementable, computationally efficient and provide a human-like reasoning structure.

On Table 1 we can find the connectives used on the current version of SNePS[5] to perform *node-based inference* (Shapiro, et al., 2004). SNIP is responsible for the implementation of each deduction rule. A more detailed description of the way a rule node produces solutions is found on section 2.3.4.

On this version of SNePS universal quantification is implemented. This is represented using a **forall** arc between the node representing the proposition and the variable node being quantified. Existential quantification is not available (although *Skolemization* of variables can be used).

Linear form	Network Representation	Deduction Rules
And-Entailment $\{A_1, \dots, A_n\}\& \Rightarrow \{C_1, \dots, C_m\}$		$\frac{A_1, \dots, A_n}{C_1, \dots, C_m}$
Or-Entailment $\{A_1, \dots, A_n\}\vee \Rightarrow \{C_1, \dots, C_m\}$		$\frac{A_i}{C_1, \dots, C_m}$, for some i, $1 \leq i \leq n$
Numerical Entailment $\{A_1, \dots, A_n\}i \Rightarrow \{C_1, \dots, C_m\}$		$\frac{A_a, \dots, A_b}{C_1, \dots, C_n}$ where $A_a, \dots, A_b$ are any i antecedents of the rule

[5] At the time of writing, the newest available implementation of SNePS is version 2.6.1.

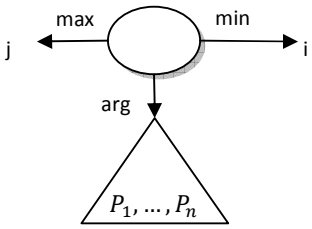
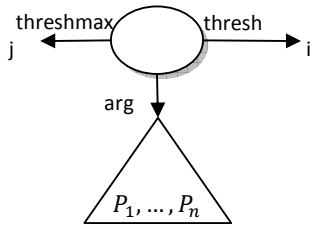
Linear form	Network Representation	Deduction Rules
And-Or $\wedge V_i^j\{P_1, \dots, P_n\}$		$\frac{P_a, \dots, P_b}{\backsim P_k, \dots, \backsim P_l}$ where $P_a, \dots, P_b$ are any j and $\sim P_k, \dots, \sim P_l$ are the remaining $n-j$ $\frac{\backsim P_a, \dots, \backsim P_b}{P_k, \dots, P_l}$ where $\sim P_a, \dots, \sim P_b$ are any $n-i$ and $P_k, \dots, P_l$ are the remaining i $\perp$: if more than j are true or more than $n-i$ are false, a contradiction is derived
Thresh $\Theta_i^j\{P_1, \dots, P_n\}$		$\frac{P_a, \dots, P_b, \backsim P_c, \dots, \backsim P_d}{P_k, \dots, P_l}$ where $P_a, \dots, P_b$ are at least i, $\backsim P_c, \dots, \backsim P_d$ are any $n-j-1$ and $P_k, \dots, P_l$ are the remaining $j-i+1$. $\frac{P_a, \dots, P_b, \backsim P_c, \dots, \backsim P_d}{\sim P_k, \dots, \sim P_l}$ where $P_a, \dots, P_b$ are any $i-1$, $\backsim P_c, \dots, \backsim P_d$ are at least $n-j$ and $\sim P_k, \dots, \sim P_l$ are the remaining $j-i+1$. $\perp$: when at least i are true and more than $n-j-1$ are false, a contradiction is derived

Table 1 – Non-standard connectives on SNePS: linear form, network representation and associated deduction rules.

2.3.3. The match algorithm

In order to locate nodes on the network that can potentially be used to derive new conclusions, either because they are asserted or because they are rule nodes, we need a unification algorithm (Shapiro, 1977). The *match algorithm* is applied to a node S and an initial *binding* β, producing a set of tuples that we will denote as $match(S, \beta)$. Each tuple has the form $\langle T, \tau, \sigma \rangle$, where T is a node, τ is a target binding and σ is a source binding.

By *binding*[6] we mean a set of the form

$$\delta = \left\{ {}^{v_1}/_{t_1}, \dots, {}^{v_n}/_{t_n} \right\},$$

where v_i are variable nodes and t_i are any nodes. We will call v_i the variable of the pair *i* and t_i the term of the same pair.

We use $N\alpha$ to denote the result of the application of the binding α to a node *N*. We will also use $R(U)$ to denote the set of nodes that are connected to node U through *arc* R.

Given the above definitions, if $\langle T, \tau, \sigma \rangle \in match(S, \beta)$, then $T\tau$ and $S\sigma$ are matched in the sense that "for every descending relation R, every atomic [constant] node in $R(T\tau)$ is also in $R(S\sigma)$ and for every [structured] node in $R(T\tau)$ is an equivalent node in $R(S\sigma)$" (Shapiro, 1977).

[6] We will use the terms *binding* and *substitution* interchangeably.

As we will see in the next section, the split of the unification into *target* and *source* bindings has a remarkable practical interest.

2.3.4. When the semantic network becomes a network of processes

Types of processes

When a deduction is initiated by the user through the command **deduce**, SNIP creates and associates a process to each node involved on the deduction (Hull, 1986). There are two major types of processes:

- *Non-rule node processes* are associated to every node not appearing on Table 1. These processes try to prove instances of the node, matching against other nodes on the network. If the matched nodes are asserted, then new solutions were found. If they are not, then SNIP tries to prove the node eventually applying rules which have that node as a consequent;[7]
- *Rule-node processes* are associated to the nodes that follow the case frames of Table 1. When these nodes are asserted and molecular (structural with no free variables[8]), they implement each deduction rule. However when the rule node has free variables and is not asserted, then it acts as an ordinary pattern node and tries to prove rule instances as a general propositional node.

The communication between processes is made through *channels.* There are three types of channels: *regular channels*, *feeders* and *consequent channels*. Channels run parallel to the arcs on the network and propagate both *requests* and *reports*.

Each process has a set of *feeders* (INCOMING-CHANNELS) and *regular channels* (OUTGOING-CHANNELS). Rule-node processes also have a set of *consequent channels* (RULE-USE-CHANNELS), which are parallel to *cq* and *arg* arcs (see Table 1).

Sending Requests

A node sends a request to another node when it is interested in its "services".[9] Requests ask for the deduction of a solution for a given node:

a) Non-rule nodes:
   i) They request solutions to other matched nodes in the network with the same structure;
   ii) They send requests to the rule nodes that have them as a consequent. Afterwards, the rule node will try to prove its antecedents, conclude the consequent and return solutions.

[7] Although it is clear for *and-entailment*, *or-entailment* and *numerical-entailment* what is meant by a node being a consequent of a rule, this is not obvious for *thresh* and *and-or* nodes. In the first case, the arc *cq* clearly indicates the consequents of the rule. However in the second, due to the nature of these rule nodes, the nodes pointed by arc *arg* can be potentially used both as antecedents and as consequents.

[8] Not having free variables means that even if the node dominates some variables, they are quantified using, for instance, a forall arc.

[9] When we say that a "rule node sends a request", we are obviously saying that a "the process associated to the rule node sends a request". Since there is a one-to-one correspondence between nodes and processes, we will assume that both expressions are equivalent and use them interchangeably.

When non-rule nodes send a request to another node on the network, an INCOMING-CHANNEL is installed waiting for reports from that node. The receptor handles the requests, installing the corresponding OUTGOING-CHANNEL. If a non-rule node is a consequent and sends a request to an asserted rule, the rule node installs a RULE-USE-CHANNEL instead.

b) Rule nodes:
    i) If the rule node is asserted and if it is asked to derive a given consequent, it requests solutions for all its antecedents;
    ii) If the rule is not asserted, then it will act as a general non-rule node, trying to infer instances of the rule.

When an asserted rule node sends a request to an antecedent, the rule node will similarly install an INCOMING-CHANNEL. The antecedent will install the corresponding OUTGOING-CHANNEL. This behavior is similar when non-asserted rule nodes try to find rule instances and act as general non-rule nodes.

Replying Reports

Reports inform the requestor that a new instance of the node was derived:

c) Non-Rule nodes:
    i) When non-rule nodes are asserted, then they are considered solutions. Therefore, they send reports through the previously installed OUTGOING-CHANNEL, containing the binding of the respective solution.
    ii) They send reports of new instances to rule nodes, which have the non-rule nodes as antecedents and have previously requested solutions;
    iii) When non-asserted non-rule nodes receive a report, for instance from a rule node that has that node as a consequent, then the non-rule node reports to the nodes that previously matched it.
d) Rule nodes:
    i) When a rule can be applied and the consequent can be inferred, the rule node sends a report to the consequents with the newly derived instances.
    ii) On the other hand, a rule node also acts as a simple propositional node. Therefore it can inform other non-asserted rule nodes of new rule instances.

The information flow on the network informally presented above, only accounts for the backward inference mechanism. The limited forward inference implemented by SNIP causes reports to be sent to nodes which have not requested them.

Putting interactions together

Now it may be interesting to follow an example of a deduction, illustrating each of the message exchanges described above. On Figure 4, Figure 5 and Figure 6 we will assume that thin black dotted arrows are message exchanges and light grey thick double arrows are channels established between nodes. Coupled to each

message exchange arrow we can find an index, which corresponds to each of the items of the four lists above. The example is divided into three phases, just for ease of presentation.

Phase 1: Let's admit that we are interested in finding instances of node Pi, which is a non-asserted pattern node[10]. The following process starts (see Figure 4):

- Pi tries to match itself on the network. We will assume that only two "matchable" nodes are present, namely Mj! and Pcq;
- Pi sends requests (of type (a)(i)) to both nodes Mj! and Pcq. Match channels are established between nodes (Pi/ Mj! and Pi/Pcq);[11]
- Mj! is asserted and corresponds to a solution;
- Pcq is not asserted and as a result it tries to prove itself requesting each of the nodes that point it as a consequent (in this case node Mr!), using requests of type (a)(ii). A consequent channel is established between Mr! and Pcq. We shall notice that Pcq does not try to match itself again on the networks, as that would cause an infinite loop of match operations.

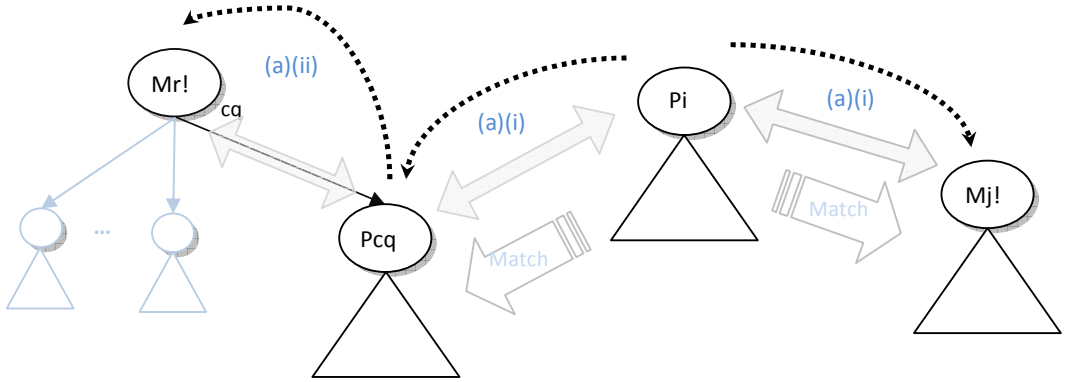


Figure 4 – First phase of the deduction of instances of Pi: Pi and Pcq send requests.

Phase 2: Now as Mr! is asserted, the rule node tries to infer the consequent in the following manner (please follow the steps on Figure 5):

- Mr! sends requests of type (b)(i) to each of the nodes on the antecedent position.[12] Communication channels are established between antecedent nodes and Mr!;
- As soon as some solutions are found to those antecedents, these are reported to Mr! using messages of type (c)(ii);
- When enough information is gathered on Mr! (see "Production of solutions on rule-nodes"), reports with solutions are sent to Pcq.

[10] It important to note that it is not syntactically possible to have a pattern node as a top node on the network. However when **deduce** is executed, a temporary pattern node is created to start the inference process.
[11] Previously we asserted that "channels run parallel to arcs". Sorry, not absolutely true!
[12] Although we used *&ant* arcs (and-entailment), this process is analogous if we had *ant* or *arg* arcs.

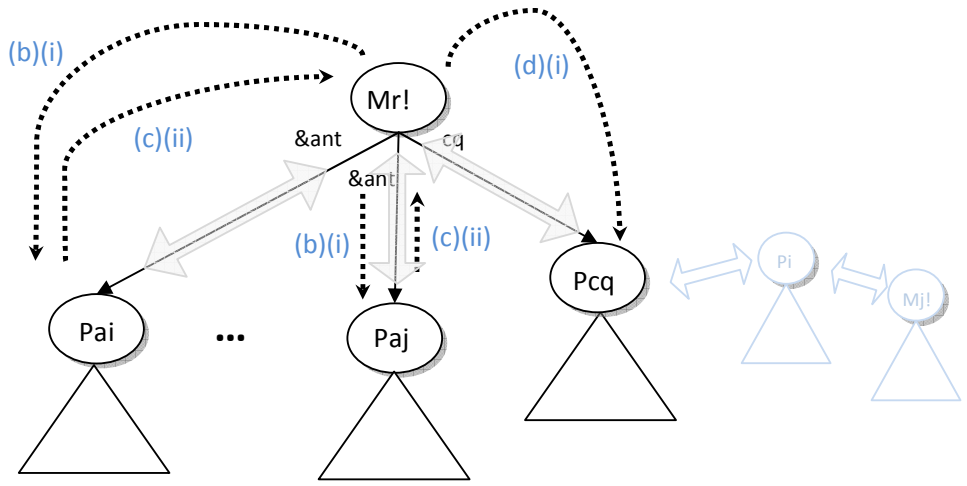


Figure 5 – Second phase of the deduction process: the rule node Mr! tries to infer instances of the consequent Pcq.

Phase 3: Finally solutions are reported to Pi (this time check Figure 6):

- Pcq sends a report message of type (c)(iii) to Pi, with the appropriate substitution derived by the rule node.
- Similarly, Mj! reports to Pi.

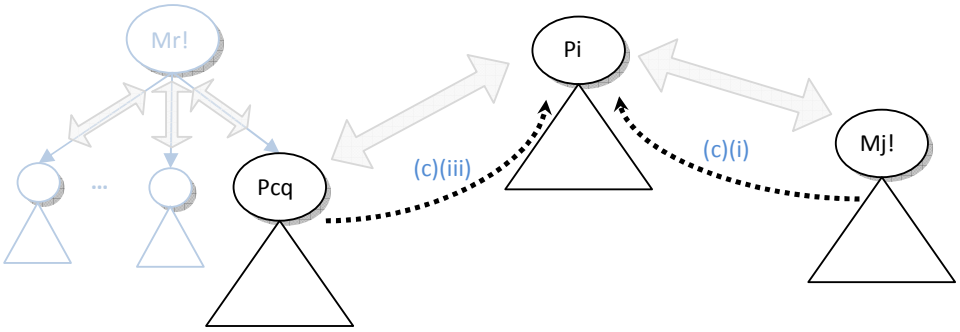


Figure 6 – Final Phase: nodes Pcq and Mj! report new instances to Pi.

After this example, interactions of type (b)(ii) and (c)(ii) remain to be shown since Mr! is asserted. However it is not difficult to conceive a non-asserted rule node sending a request through a consequent channel, with the intention of asking for asserted instances of the rule.

Producer-consumer model

All the processes are *data collectors* (Shapiro, et al., 1980). This means that:

- they store all the solutions found to the node they are associated to (the process register KNOWN-INSTANCES is used for that purpose);

- each data collector has a series of *bosses* to which it reports newly discovered instances, through the use of the OUTGOING-CHANNELS;
- they never produce the same solution instance twice to the same "boss";
- when a new boss is attached to a data collectors, that boss immediately receives all the already KNOWN-INSTANCES.

To the *requestor* we will call *consumer* (since it consumes the solutions produced). To the node that provides the "service" to the boss we will call *producer* (since it produces solutions to the requestor).

Filtering solutions, switching contexts

Taking into account the above description, it is now possible to understand the importance of having both *source* and *target bindings* rather than a single unification. As we showed before, when a given node matches another node and sends a request for known instances, the two nodes create a channel between each other. The first node (*consumer*) becomes the *boss* of the second (*producer*) and waits for reports of new solutions. The same node can report to more than a single *consumer*. Two consequences derive from this fact:

- A node may produce solutions that are needed by one boss, but not by another. Therefore, some solutions may need to be *filtered out* because they may be too generic.
- When a certain node sends a report containing bindings with new solutions, these bindings are expressed in terms of the variables of that node. As a consequence, the space of the variables of the *producer* must be *switched* to the space of the *consumer*.

The first task is accomplished by the target binding (from now on called *filter*). The second is performed by the source binding (now called *switch*).

The example of Figure 7 will make these concepts clear. Let's admit that the match algorithm is applied to a temporary node P6 and that the network contains the *or-entailment* node (a).

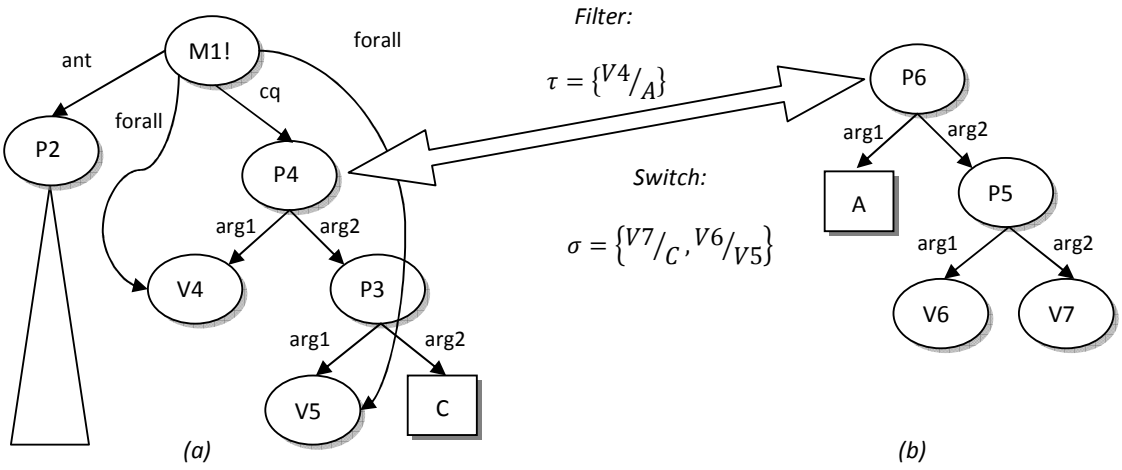


Figure 7 – The filter and switch bindings when node P6 and P4 are matched.

The result of applying $match(P6, \{\ \})$ would be $\langle P4, \{{}^{V4}/_{A}\}, \{{}^{V7}/_{C}, {}^{V6}/_{V5}\}\rangle$. Now, let's admit also the following sequence of interactions:

1. P6 send a request to P4. That request establishes a channel between P6 and P4 (indicated by the big arrow);
2. P4 tries to prove itself sending a request to the rule node M1! through a channel parallel to *cq*;
3. M1! sends requests to the antecedent nodes represented generically by a triangle.
4. M1! receives reports from its antecedents and draws two conclusions that are sent on a report to P4. The two instances found are $\alpha = \{{}^{V4}/_{A}, {}^{V5}/_{B}\}$ and $\beta = \{{}^{V4}/_{B}, {}^{V5}/_{D}\}$.

Considering this scenario, it is easy to see that only the first solution is interesting to P6. On the second solution, V4 is bound to the constant B and the corresponding node in P6 is the constant A. Therefore β is filtered out, as the pair ${}^{V4}/_{A}$ is present on the filter binding. If a solution binds a variable to a term that is not compatible to the term that variable is bound on the filter, then that solution should not be reported to the requestor.

On the other hand the solution $\alpha = \{{}^{V4}/_{A}, {}^{V5}/_{B}\}$ is expressed in terms of the variables of the pattern node *P4*. If node *P6* receives α, it cannot assign any meaning to that solution since *V4* and *V5* are nodes from *P4*. Besides this the solution assumes $arg2(P3) = C$, and clearly this does not happen in the corresponding node $arg2(P5)$. The switch of variable contexts is made through the source binding σ. The *consumer* should receive $\sigma\backslash\alpha$[13], that is the *switch* σ composed with α. In our example $\sigma\backslash\alpha = \{{}^{V7}/_{C}, {}^{V6}/_{B}\}$.

Production of solutions on rule-nodes

From the last column of Table 1, it is obvious that both positive and negative premises can be used by rule nodes to perform inference. Besides that rule nodes can also derive a negative conclusion, which means that they can prove that a given instantiation of a node is false.

Taking this fact into account, solutions to a node must contain a binding and a sign specifying whether that binding represents a positive or negative instance of the node.

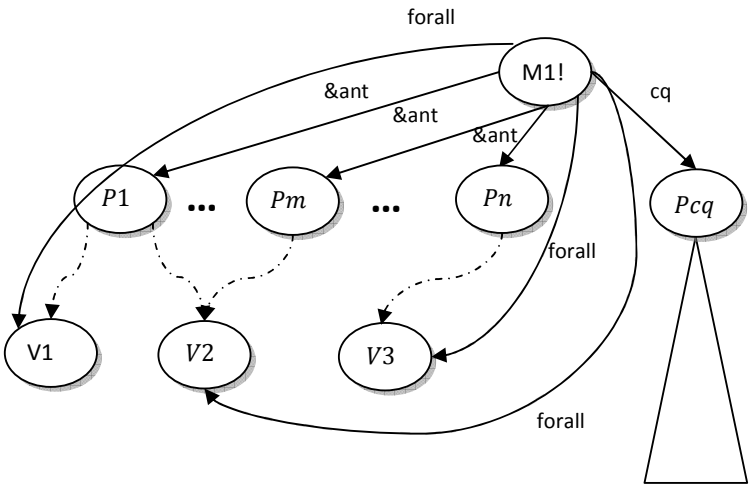
It is clear that every rule node must maintain a list with all the reported instances of its antecedents. When enough information is gathered about the antecedents, inference takes place. The number of proved antecedents needed varies obviously from rule node to rule node.

When a rule node sends requests to its antecedents and receives reports back, the substitutions found to each universally quantified variable are combined. The way this combination is made depends on:

[13] The composition $\sigma\backslash\alpha$ is calculated substituting each pair ${}^{v}/_{t1} \in \sigma$, for which *t1* is a variable on some pair ${}^{t1}/_{t2} \in \alpha$, by ${}^{v}/_{t2}$.

- the type of the rule node: while conjunctive connectives like and-entailment need solutions to all the antecedents, or-entailment needs to prove only one of them (each positive report from an antecedent, immediately gives rise to a solution to the consequents);
- the way common variables are shared among antecedents: if antecedent nodes are independent in the sense that they don't share variables, then we can combine the different solution bindings by simply taking their union. On the other hand when variables nodes are shared, then we must assure that the terms assigned to common variables are compatible.

Figure 8 shows a network representing an *and-entailment* node with *n* antecedent nodes. The dotted arrows mean that *Pi* dominates some variable *Vj*.



$s_{1,1} = \{V1/_A, V2/_B\}$

$s_{1,2} = \{V1/_A, V2/_C\}$

$s_{1,m} = \{V2/_C\}$

$s_{2,m} = \{V2/_F\}$

$s_{1,n} = \{V3/_D\}$

$s_{2,n} = \{V3/_E\}$

Figure 8 – Combining substitutions from different antecedents.

If node *P1* discovers solutions $s_{1,1}$ and $s_{1,2}$, if node *Pm* discovers $s_{1,m}$ and $s_{2,m}$ and *Pn* discovers $s_{1,n}$ and $s_{2,n}$ then only two solutions would obviously be propagated as reports to consequents: $\{V1/_A, V2/_C, V3/_D\}$ and $\{V1/_A, V2/_C, V3/_E\}$.

However if instead of having an *and-entailment* we had an *or-entailment* rule, then each of the six different solutions reported by the antecedents would produce a different report to the consequents.

The way solutions are stored on rule nodes can be uniformly handled using the *rule use information* (*RUI*) data structure for all the node types. RUI data structures are a 4-tuple of the form $(\langle substitution \rangle, \langle non-negative\ integer \rangle, \langle non-negative\ integer \rangle, \langle flagged-node\ set \rangle)$. The first integer is the number of positive antecedents and the second the number of negative antecedents found for the given substitution. $\langle flagged-node\ set \rangle$ is a list that for each antecedent reported specifies whether a positive or negative instance was derived.

On our example and assuming that M1! is an *and-entailment* node, we would have for the first solution $\left(\{{}^{V1}/_{A}, {}^{V2}/_{C}, {}^{V3}/_{D}\}, 3, 0, \{P1: true, P2: true, P3: true\}\right)$. In the case of the *or-entailment* we would obtain $\left(\{{}^{V1}/_{A}, {}^{V2}/_{B}\}, 1, 0, \{P1: true\}\right)$ as soon as the first report from P1 would arrive.

It is interesting to note that, for every node on Table 1, the possible derivation of new conclusions can be determined only in terms of the two integers on the RUI. For instance in case we have a numerical *entailment* node $\{A_1, \ldots, A_n\}i \Rightarrow \{C_1, \ldots, C_m\}$, then when the first integer of some RUI is equal to *i*, a new conclusion can be inferred. The *and-or* rule node is an example of a node that needs information from both integers, since inference can be made based on false antecedents.

(Choi, et al., 1992) shows how RUI data structures help to control the combinatorial explosion that arises when we combine substitutions from the different antecedents and how the appropriate data structure depends on the conjunctive and non-conjunctive nature of the connectives.

2.3.5. Deciding which process gets into action – the MULTI system

MULTI is a LISP based multiprocessing system which allows "various control structures such as recursion, backtracking, coroutines and generators" (McKay, et al., 1980).

MULTI implements:

- a set of primitives which allows the creation of processes and the manipulation of their registers;
- an evaluator which runs on two modes: single or double queue. On single queue mode, the evaluator continually takes the process on the top of the queue and evaluates it until no more processes remain. Queues on double queue mode have different priorities: the low priority queue is only processed when no processes remain on the high priority queue.
- a scheduler which decides on the type of scheduling policy applied to each queue.

On SNePS, the evaluator is generally used on double queue mode. This way processes that receive reports of new instances are put on the high priority queue to assure a faster propagation of solutions on the network.

The scheduler implements a FIFO policy. New network processes are added to the end of the queue, giving rise to a breath-first search.

Table 2 shows the set of registers of each process, some of which were already referred before.

Register	Description
NAME	Type of process: NON-RULE, RULE or USER.
TYPE	Used on rule nodes only. Indicates the type of process, according to the type of connective of the node (e.g. AND-ENTAILMENT, AND-OR etc).
NODE	The node the process is associated to (e.g. V3, M2!, P1).
KNOWN-INSTANCES	The instances derived for this node, with their respective substitution, context of derivation, and sign (POS or NEG)
REPORTS	The set of reports received, containing the substitution, the sign of the solution (whether the solution is true or false) and the signature of the producer of the report.

Register	Description
REQUESTS	The set of requests received. Requests are implemented as channels that are installed by the producer node as an OUTGOING-CHANNEL.
OUTGOING-CHANNELS	Set of *channels* the node will use to report solutions. The *channel* contains a *filter* and a *switch* as described on section "Filtering solutions, switching contexts". It also contains a *valve* that can be used to control inference (can be closed or open, allowing reports to be sent or blocked), and the name of the *destination* node.
INCOMING-CHANNELS	Set of *feeders* that will bring *reports* of newly discovered instances of nodes. Its structure is similar to the channel.
RULE-USE-CHANNELS	Set of *consequent channels*. Each of these channels contains the channel used to report solutions to the consequent, the set of antecedent nodes and a set of RUI structures as described on section "Production of solutions on rule-nodes".
PENDING-FORWARD-INFERENCES	Set of reports that will be sent to perform forward inference.
PRIORITY	Assumes two possible values, LOW or HIGH, according to the queue the process was scheduled on MULTI.
RULE-HANDLER	Used on rule nodes only. Name of the LISP function in charge of producing new consequents of the rule, given the instances derived from the antecedents.
USABILITY-TEST	Used on rule nodes only. Name of the LISP function that given the sign of the rule node, decides whether the rule is applicable to derive new conclusions or not.

Table 2 – Set of registers of each process.

When the user asks for a deduction using the command **deduce**, a user process is created. This user process stores in its registers the number of positive and negative instances found and desired by the user. This special process, which is an exception to the fact that every process is connected to a node on the network, stores all the deduced nodes and returns them back to the user as soon as the inference process is ready. This process can be seen as the *boss* of every process on the network.

At each moment of the inference process, a process can be *scheduled* or *idle*. While a *scheduled* node is waiting on some queue to be executed, an *idle* process may be waiting for reports of other nodes or simply may not have been requested to infer any solutions. Therefore it is not present in any queue.

When process *p1* sends a request to process *p2*, then the consumer p1 is in charge of scheduling the producer p2 in the low priority queue. When p2 returns reports to p1, then it is necessary that p2 inserts p1 on the queue, this time on the high priority queue, assuring the quicker propagation of solutions.

3. The resolution mechanism

Now that we have an idea of how SNePS performs inference, we turn to one of the most important automatic deduction mechanisms: **resolution**.

Our incursion on this field is not certainly innocent. We believe that it is crucial to establish a comparison between both methods for two main reasons that we now explain. First, because this exercise will give us a greater insight of the potentials and drawbacks of SNIP. Secondly because we hope to find interesting concepts and ideas that could be tested on SNIP with the intention of making the deduction process more efficient.

The choice of *resolution* was not arbitrary. Since its first presentation by its creator on (Robinson, 1965), many authors contributed to the method suggesting improvements. Its simplicity, power and computational adequacy made it resist across time and nowadays it is still used as the basis of award winning theorem provers like Prover9, the successor of OTTER (McCune, et al., 1997).

In this section, we will first be interested in understanding the basic resolution method and giving an intuition of the theoretical results upon which the mechanism is based. Afterwards we will go through some resolution refinements, trying to understand how these modifications improved the method and paying attention to the concepts behind these improvements. We will focus on *SLD-resolution*.

3.1. Introducing Resolution

The aim of any first-order logic inference system is to, starting from a set of formulas Γ, find a proof for a formula φ applying a sequence of *inference rules*. The existence of a such a derivation is represented by $\Gamma \vdash \varphi$.

If the system is *sound* and $\Gamma \vdash \varphi$, then we can infer that φ is a semantic consequence of Γ. Semantic consequence is represented by $\Gamma \vDash \varphi$.

Resolution reformulates the problem of showing $\Gamma \vDash \varphi$, into the proof of the *unsatisfiability* of the set $\Gamma' = \Gamma \cup \{\backsim \varphi\}$.[14] A set Δ is said to be unsatisfiable if it has no models. Therefore *resolution* presents a method that seeks for the *refutation* of a set of clauses.

3.1.1. Preparing formulas for Γ': from first-order formulas to clauses

The *resolution rule* works with clauses and therefore we will start by claiming the existence of set of transformations that accomplishes the translation from first-order formulas to clauses.

A *clause* is a set of *literals* and a *literal* is an *atom* or the *negation of an atom*. An *atom* is a propositional symbol applied to terms. If P is an atom, we will represent the negation of that atom as $\sim P$. P and $\sim P$ are said to be *complementary literals*.

An example of a clause could be

[14] This equivalence can be easily shown.

$$C = \{P(x, a), \sim Q(b, f(x), y)\}.$$

Each *clause* is semantically equivalent to a first-order formula where all the variables are universally quantified and the literals are disjunct. Hence the previous clause is equivalent to

$$\forall x \forall y \big(P(x, a) \vee \sim Q(b, f(x), y)\big).$$

The transformation enunciated above can be accomplished in four steps:

- In the first, we use a *rewrite system* that transforms a first order formula into the *prenex normal form*. This form is $Qx_1, \dots, Qx_n \psi$ where:
    1. Qx_i is the universal or existential quantification of variable x_i (that is Q stands for ∀ or ∃);
    2. ψ is the *matrix* of the formula and is quantifier free.

    We also impose that no free variables exist on the formula. This rewrite system and the one presented in the bullet below are presented, for instance, on (Socher-Ambrosius, et al., 1997).

- The second step is to transform the *matrix* ψ of the resulting formula into the *conjunctive normal form*, that is in a formula of the type

    $$\psi' = \Big((\alpha_{11} \vee \dots \vee \alpha_{1i}) \wedge \dots \wedge \big(\alpha_{j1} \vee \dots \vee \alpha_{jk}\big)\Big),$$

    where α_{kl} are literals. Again, there is a rewrite system that accomplishes this task.

- Next we *skolemize* existentially quantified variables, so that we get rid of existential quantifiers. Existentially quantified variables are substituted by a new function (called *skolem function*) applied to all the variables previously universally quantified. This way,

    $$\forall x_1, \dots, \forall x_{i-1} \exists x_i, \dots, Qx_n \psi$$

    is transformed into

    $$\forall x_1, \dots, \forall x_{i-1}, \dots, Qx_n \, \psi \left\{ {}^{x_i}/_{f_{x_i}(x_1, \dots, x_{i-1})} \right\},$$

    where $\psi \left\{ {}^{x_i}/_{f_{x_i}(x_1, \dots, x_{i-1})} \right\}$ is the substitution on ψ of all the occurrences of x_i by $f_{x_i}(x_1, \dots, x_{i-1})$. This process is repeated until no more existential connectives are present on the formula, which is now in the *Skolem conjunctive formula*.

- Finally we drop the universal quantifiers (all the variables will be implicitly universally quantified), and from the final formula[15]

    $$(\alpha'_{11} \vee \dots \vee \alpha'_{1i}) \wedge \dots \wedge \big(\alpha'_{j1} \vee \dots \vee \alpha'_{jk}\big),$$

    we obtain *j* sets of literals

    $$\{\alpha'_{11}, \dots, \alpha'_{1i}\}, \dots, \{\alpha'_{j1}, \dots, \alpha'_{jk}\}.$$

The sequence of transformations described above is sketched on Figure 9.

[15] Note that α_{ij} is substituted by α'_{ij} since eventually some variables were skolemized.

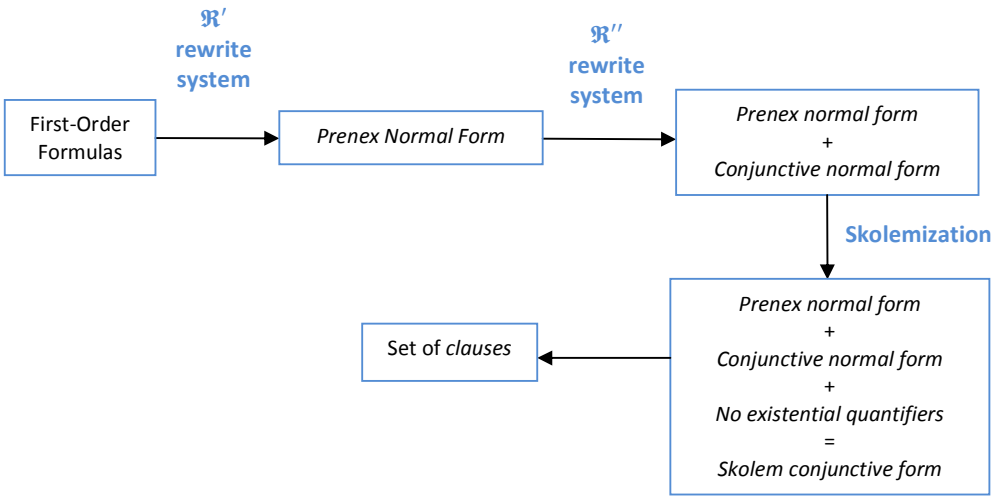


Figure 9 – Transformation of first order logic formulas into clauses.

The set of formulas obtained after the first two steps is related to the initial set of formulas by the logical relation of equivalence. The last step is merely a "cosmetic" change. Skolemization preserves *satisfiability*, that is formulas before step three are satisfiable if and only if formulas after that step are also satisfiable. Consequently, the set of clauses obtained is satisfiable iff the original set of first-order formulas is also satisfiable.[16]

However this loss of equivalence does not compromise our task. Remember that we are interested in showing the unsatisfiability of the set Γ' and these transformations do not change that property.

3.1.2. The Herbrand Theorem

We now turn to the problem of proving the unsatisfiability of a formula in *Skolem conjunctive form*. The *Herbrand theorem* is a central result that can be used for this purpose. We cite the version presented on (Loveland, 1978), p. 49: "A formula A in Skolem conjunctive form is unsatisfiable iff there exists a finite set of ground clauses[17] of A which is unsatisfiable".

We give a brief example: let´s admit the formula

$$A = \forall x \left(\left(\sim A\big(f(x)\big) \vee B(x) \right) \wedge \left(A(x) \right) \wedge \left(\sim B(a) \right) \right).$$

We remark that A is in *conjunctive normal form*, that the formula is unsatisfiable, and that we could translate A into three clauses

$$A' = \Big\{ \big\{ \sim A\big(f(x)\big), B(x) \big\}, \{A(x)\}, \{\sim B(a)\} \Big\}.$$

According to the theorem above, to prove the unsatisfiability of A, we present the finite set of ground clauses

[16] Or equivalently, the set of clauses obtained is **un**satisfiable, iff the set of first-order formulas is also **un**satisfiable.
[17] A ground clause is a clause where all variables are bound to some constant.

$$A'' = \{\{\sim A(f(a)), B(a)\}, \{A(f(a))\}, \{\sim B(a)\}\}.$$

This set is unsatisfiable because given that $A(f(a))$ and $\sim B(a)$ need to be both true, the clause $\{\sim A(f(a)), B(a)\}$ is consequently false. Hence A is unsatisfiable using the previous theorem. The *Davis-Putnam procedure* is particularly useful to algorithmically prove the inconsistency of a set of ground clauses (Davis, et al., 1960).

As suggested by (Robinson, 1965) and based on the reasoning above, we could immediately devise a *saturation procedure* for solving our initial problem. The intuition for this procedure is simple: "if we generate all possible sets of ground clauses for a given *conjunctive normal* formula and if we use the *Davis-Putnam* algorithm to check their consistency then, if it exists, we will certainly come across an inconsistent set. If it doesn't and the set of possible ground clauses is infinite, we will continue forever". To better understand this odd approach we will follow the steps below:

- First we define the *Herbrand universe* as the (finite or infinite) set containing all the possible terms that could be built with the constants and functions of the language. For instance the *Herbrand universe* of the set of clauses A' is defined as the infinite set
$$\mathcal{U}_C = \{a, f(a), f(f(a)), \dots, f^i(a), \dots\}.$$ [18]
- Then we build a sequence of subsets $P_1, P_2, \dots, P_i, \dots$ of the set $\mathcal{U}_C$, such that $P_i \subset P_{i+1} \subset \mathcal{U}_C$ and also that $\mathcal{U}_C = \bigcup_{i=1}^{\infty} P_i$. On our example, we could define $P_i = \{f^j(a) | 0 \leq j \leq i\}$.
- Next we define the *Herbrand base* as the *saturation* of a set of clauses with the *Herbrand universe*. The *saturation* of a set of clauses S given a set of terms P is written $P(S)$. It returns all possible ground clauses that could be obtained instantiating all the variables of the clauses of the set S, with the terms of the set P. Going back to our example,
$$P_i(A') = \left\{ \begin{matrix} \{\sim B(a)\}, \\ \{A(a)\}, \{A(f(a))\}, \dots, \{A(f^i(a))\}, \\ \{\sim A(f(a)), B(a)\}, \{\sim A(f^2(a)), B(f(a))\}, \dots, \{\sim A(f^i(a)), B(f^{i-1}(a))\} \end{matrix} \right\}.$$
- Then we could check each $P_i(A')$ for satisfiability using the *Davis-Putnam procedure*. On our example, $A'' \subset P_2(A') = \{\{\sim B(a)\}, \{A(a)\}, \{A(f(a))\}, \{\sim A(f(a)), B(a)\}\}$ is unsatisfiable.

Although this algorithm is effective, it is computationally very inefficient. Robinson's innovation was to find a procedure that guides the creation of the sets P_i.

3.1.3. Ground Resolution

To introduce the more general principle of *resolution*, we present a restricted form of resolution called *ground resolution*, which has the same purpose as Davis and Putnam's work and works only with ground clauses.

[18] $f^i(a)$ denotes the function f applied i times to the constant a.

Ground resolution is an inference system that contains one single *sound inference rule* called *ground resolution rule*. This rule receives two ground clauses $C_1 = \{L_1, \dots, L_i\}$ and $C_2 = \{L'_1, \dots, L'_j\}$ and two **complementary** ground literals $L_k \in C_1$ and $L'_l \in C_2$. It produces a third clause, which is a semantic consequence of the previous two. The rule can be written in the following way:

$$R(C_1, C_2, L_k, L'_l) = (C_1 \cup C_2) - \{L_k, L'_l\}.$$

A *deduction* of the ground clause C from a set of ground clauses S is a sequence of ground clauses $D = (C_1, C_2, \dots, C_n)$ such that:

- $C = C_n$;
- each $C_i \in D$ is obtained either:
    1. $C_i \in S$;
    2. applying the ground resolution rule to C_j and C_k such that $j, k < i$.

We write $S \vdash_{gR} C$ to denote the existence of a derivation of the clause C from the set S.

The empty clause is denoted by □ and is named *contradiction*. This means that if we show $S \vdash_{gR} \square$, then we find a *deduction* called *refutation*. A *refutation* makes an implicit *contradiction*, explicit. Consequently due to the soundness of the inference rule, we showed that the set S is unsatisfiable.

As an example, on Figure 10 we show that $P_2(A')$ is unsatisfiable proving that $P_2(A') \vdash_{gR} \square$. The Davis-Putnam procedure referred above is no longer necessary to prove the unsatisfiability of sets of clauses.

	Justification of steps:
$D_1 = \{A(f(a))\}$	$D_1 \in P_2(A')$
$D_2 = \{\sim A(f(a)), B(a)\}$	$D_2 \in P_2(A')$
$D_3 = \{\sim B(a)\}$	$D_3 \in P_2(A')$
$D_4 = \{B(a)\}$	$D_4 = R\left(D_1, D_2, A(f(a)), \sim A(f(a))\right)$
$D_5 = \square$	$D_5 = R(D_3, D_4, \sim B(a), B(a))$

Figure 10 – Since there exists a ground deduction $D = (D_1, D_2, D_3, D_4, \square)$, then $P_2(A') \vdash_{gR} \square$. Therefore, $P_2(A')$ is unsatisfiable.

3.1.4. General Resolution

Although the simplicity of the above inference system is fascinating, *ground resolution* by itself introduces no innovation over Davis-Putnam's method. What makes Robinson's work remarkable was the ability to ***lift*** ground resolution to first-order logic.

For that purpose Robinson devised a *unification* algorithm for first-order formulas. This algorithm receives a set of positive literals $A = \{P_1, \dots, P_m\}$ and returns a *substitution* $\theta = \left\{{}^{v_1}/_{t_1}, \dots, {}^{v_n}/_{t_n}\right\}$, where $v_1, \dots, v_n$ are variables, $t_1, \dots, t_n$ are terms and $v_i \neq t_i$for all i. The *instantiation* of a literal P_i by the substitution θ is denoted by $P_i\theta$ and is obtained by the **simultaneous** replacement of every occurrence of each variable by the respective term. The *substitution* returned by the *unification algorithm* is called a *unifier* and has the property that $P_i\theta = P_j\theta$ for every $i, j \in \{1, \dots, m\}$, that is $\#\,(A\theta) = 1$.[19] Returning to the example presented before and considering $A = \{A(f(x_1)), A(x_2)\}$, then $unify(A) = \left\{{}^{x_2}/_{f(x_1)}\right\}$, since $\#\left(A\left\{{}^{x_2}/_{f(x_1)}\right\}\right) = \#\,\{A(f(x_1))\} = 1$.

Besides the introduction of a *unification algorithm*, Robinson reformulated the *resolution rule*. Let both literals L_k and $L'_l = \sim L''_l$ be *potentially complementary literals* such that $\theta = unify(\{L_k, L''_l\})$. Then,

$$R(C_1, C_2, L_k, L'_l) = \big((C_1 \cup C_2) - \{L_k, L'_l\}\big)\theta.$$[20]

Finally and to assure the completeness of the system, we need to include the *factoring* rule: if $L_k, L_l \in C_m$ and $\theta = unify(\{L_k, L_l\})$, then

$$F(C_m, L_k, L_l) = C_m\theta.$$

We are now capable of rewriting the definition of *deduction* presented above for ground resolution: a *deduction* of the clause C from a set of clauses S is a sequence of clauses $D = (C_1, C_2, \dots, C_n)$ such that:

- $C = C_n$;
- each $C_i \in D$ is obtained through one of the following:
    1. $C_i \in S$;
    2. applying the resolution rule to C_j and C_k such that $j, k < i$;
    3. factoring some clause C_l such that $l < i$.

On Figure 11 we use the previous example again, this time to illustrate the refutation of the set A' using general resolution.

[19] $A\theta$ means the instantiation of every element of the set A. $\#\,(S)$ means the cardinality of the set S, that is the number of elements of the set.

[20] One simplification was made on this presentation: we are assuming that C_1 and C_2 do not share variables. If that is not the case then we need to standardize variables apart before unifying clauses.

	Justification of steps:
$D_1 = \{A(x)\}$	$D_1 \in A'$
$D_2 = \{\sim A(f(x)), B(x)\}$	$D_2 \in A'$
$D_3 = \{\sim B(a)\}$	$D_3 \in A'$
$D_4 = \{\sim A(f(a))\}$	$D_4 = R(D_2, D_3, B(x), \sim B(a)), \theta = \{^x/_a\}$
$D_5 = \square$	$D_5 = R\left(D_1, D_4, A(x), \sim A(f(a))\right), \theta = \left\{^x/_{f(a)}\right\}$

Figure 11 – Refutation of the set A' using general resolution.

3.1.5. A level saturation search procedure

The previous section presents an *inference system* with two *inference rules* (*factoring* and *resolution*). However if our intention is to implement *resolution* on a computer such that automatic reasoning can be made, we need to establish a search strategy: "a *proof procedure* consists of an *inference system* supplemented by a *search strategy*" (Kowalski, et al., 1983).

The *search space* generated for some set *S* is the set of all clauses obtained applying inference rules to elements of set *S* and to all other generated clauses. A *search space* can be depicted on a tree-like diagram, like Figure 12 shows, where arrows with the same numbers indicate resolution operations.

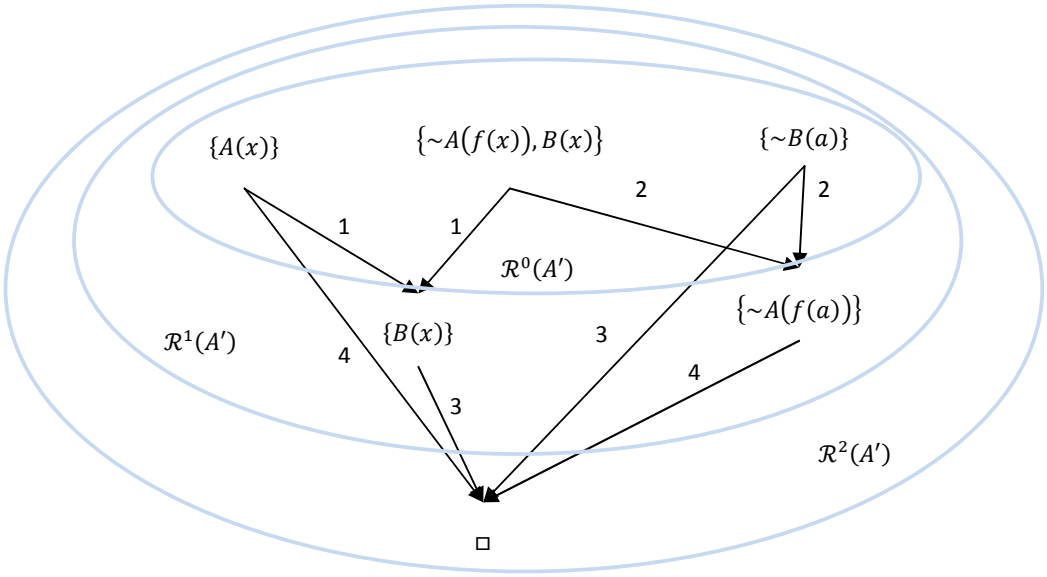


Figure 12 – Search space obtained for the refutation of set $A' = \{\{\sim A(f(x)), B(x)\}, \{A(x)\}, \{\sim B(a)\}\}$ through general resolution.

A *search strategy* guides the selection of the next inference rule to be applied as well as the input clauses for that rule, defining the way the *search space* is traversed. A *fair search strategy* guarantees that all the clauses of the *search space* are generated.

It is interesting to notice that, on example of Figure 12, there are two possible *refutations* for the initial set. As a consequence, two different search strategies could produce two different deductions.

The *level saturation search strategy* induces a *breath-first search* on the *search space.* Let $\mathcal{R}(S)$ denote the set that results from the application of resolution and factoring to all pairs of clauses of S, that is $\mathcal{R}(S) = S \cup R\left(C_i, C_j, l, \sim l\right) \cup F(C_k, l', l'')$ where $C_i, C_j, C_k \in S$ (see Figure 12). Then we can define the *level saturation strategy* by induction as $\mathcal{R}^0(S) = S$ and $\mathcal{R}^{n+1}(S) = \mathcal{R}\left(\mathcal{R}^n(S)\right)$. Each level of a *level saturation search tree* contains clauses such that $level_{i+1} = \mathcal{R}^{n+1}(S) - \mathcal{R}^n(S)$.

We finish this introduction to resolution stating Robinson's *Resolution Theorem*, presented on (Robinson, 1965): if S is any finite set of clauses, then S is unsatisfiable if and only if $\mathcal{R}^n(S)$ contains □, for some $n \geq 0$. This theorem is a reformulation of the *Herbrand Theorem* presented before and has a central role on the development of the theory of resolution.

3.2. Resolution Refinements

After Robinson's paper, many improvements were suggested to basic resolution in order to avoid the exhaustive *level saturation search* of the entire search space. According to (Loveland, 1978), the ideas proposed by many authors as *refinement* to *resolution* tried to do one of the following:

- to reorder the sequence of *resolution* or *factoring* operations applied on a deduction. This kind of refinement is called a *strategy*;
- to prune the search space, preferably not affecting the (refutation) completeness of the system.

3.2.1. Linear Resolution

Loveland proved that all *refutations* could be made *linear*. This means that after choosing an initial clause C' belonging to the unsatisfiable set *S* (satisfying certain properties we will see latter), then the empty clause can still be derived even if we impose the following three restrictions on the *resolution* and *factoring* operations:

1. that the first inference rule applied has the initial clause as a parent;
2. that at least one of the parents of each resolution operation is the clause derived in the immediately previous step (or a factor of that clause);
3. that the parent of each factoring operation is also the clause derived in the previous step.

The condition imposed on the initial clause is that there exists some *refutation* where that clause participates. If this happens then it is possible, applying a series of transformations on that refutation, to transform it in a linear one beginning with the desired clause.

Figure 13 presents a linear refutation of set A', with start clause $\{A(x)\}$. We stress the fact that the *search space* explored is smaller than the one explored using a *level saturation strategy* and that $\{\sim A(f(a))\}$ cannot be derived since none of the parents belong to the set circled in blue (the symbol "⃠" indicates that).

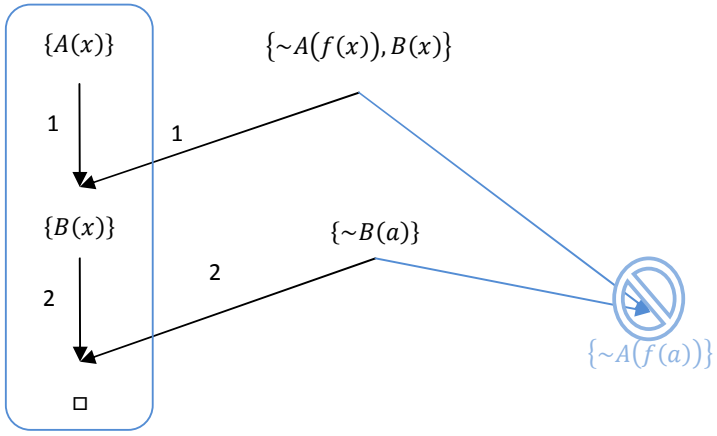


Figure 13 – Linear refutation of the set $A' = \{\{\sim A(f(x)), B(x)\}, \{A(x)\}, \{\sim B(a)\}\}$, with initial clause $\{A(x)\}$.

(Loveland, 1978) defines a *linear deduction* of the clause C from a set of clauses S as a sequence of clauses $D = (C_1, C_2, \dots, C_n)$ such that:

- $C = C_n$;
- $C_1, C_2, \dots, C_r$ is each of the r clauses of the set S;
- each $C_i \in D, i > r$ is obtained through one of the following:
    1. applying the resolution rule to C_{i-1} and a factor of C_k such that $k < i$;
    2. factoring the clause C_{i-1}.

Therefore the refutation on Figure 13 is $D = (\{\sim A(f(x)), B(x)\}, \{\sim B(a)\}, A(x), B(x), \square)$.

3.2.2. Resolution with Selection Function

Another way to restrict the number of resolution operations that can be applied to a certain clause is to include a selection function ψ. This function receives a clause and returns a literal (positive or negative) of that clause, that is $\psi(C) = l$ such that $l \in C$. A resolution operations $R(C_i, C_j, l, \sim l)$ can only be applied if $\psi(C_i) = l$.

3.2.3. SLD-Resolution

SLD-resolution combines the refinements described on sections 3.2.1 and 3.2.2, with a restriction on the type of allowed clauses. The name "SLD-resolution" means ***L**inear resolution* with ***S**election function* and ***D**efinite clauses*.

D*efinite clauses* (also called Horn clauses) have at most one positive literal:

$$C = \{\sim L_1, \sim L_2, \dots, \sim L_n, L\}.$$

The notation used generally changes to the more intuitive implicative form, which is obviously equivalent to the previous one,

$$C = L \leftarrow L_1, L_2, \dots, L_n,$$

where L is the head (the consequent) of the clause and $L_1, L_2, \dots, L_n$ is the tail of the clause (the antecedent, which is a conjunction of literals).

To restrict first order logic to the causal fragment containing only Horn clauses has a great interest since a procedural interpretation can be assigned to that fragment.

Procedural Interpretation of clauses

It is possible to look to a set of *clauses* S, which share the same head *L,* as the definition of a *procedure*. If this is the case, variables of the head *L* are parameters of the procedure and each L_i can be seen as an instruction. Thus we can define a *logical program* P as a set of Horn clauses.

As an example we could devise a rather simple program, $\mathrm{P}_1 = \{\mathrm{C}_1, \mathrm{C}_2\}$, that adds two numbers (Figure 14). We will admit that $\mathrm{s(x)}$ is a successor function, that 0 is the only constant and stands for the natural number "zero", and finally that the predicate $\mathrm{Sum(x, y, z)}$ returns on z the sum of x and y.

$$C_1 = Sum(x, 0, x) \leftarrow$$

$$C_2 = Sum(x, s(y), z) \leftarrow Sum(s(x), y, z)$$

$$C_3 = \leftarrow Sum\big(s(0), s\big(s(0)\big), z\big)$$

Figure 14 – The program $\mathrm{P}_1 = \{\mathrm{C}_1, \mathrm{C}_2\}$ adds two numbers. C_3 is a query that adds 1 and 2.

The *execution of a logical program* P can be seen as a derivation of a clause C, such that $\mathrm{P} \vDash \exists \mathrm{C}$.[21] The example above can be used to clarify this statement. If we want to add 1 and 2, we are interested in knowing if there exists any number z such that $\mathrm{P}_1 \vDash \exists \mathrm{z}\, \mathrm{Sum}\big(\mathrm{s}(0), \mathrm{s}\big(\mathrm{s}(0)\big), \mathrm{z}\big)$. Again we use SLD-resolution to refute the set $P_1 \cup \{\sim\exists z\, Sum\big(s(0), s\big(s(0)\big), z\big)\}$. But $\sim\exists z\, Sum\big(s(0), s\big(s(0)\big), z\big)$ is equivalent to the clause $C_3 = \leftarrow Sum\big(s(0), s\big(s(0)\big), z\big)$.[22] Because of this, clauses of the form

$$\leftarrow L_1, L_2, \dots, L_n$$[23]

are called *queries* or alternatively *goals*.

The *resolution* of a *goal* $G = \leftarrow L_1, \dots, L_k, \dots, L_n$ with a *clause* of the program $C = L' \leftarrow L'_1, \dots, L'_n$ gets a particularly interesting form. If θ unifies L' with L_k then

$$R(G, C, L_k, L') = (\leftarrow L_1, \dots, L'_1, \dots, L'_n, \dots, L_n)\theta,$$

[21] $\exists$C is the existential closure of the clause C, i.e if $freevars(\mathrm{C}) = \{\mathrm{x}_1, \dots, \mathrm{x}_n\}$, then$\exists$C means$\exists \mathrm{x}_1, \dots, \mathrm{x}_n$C.
[22] $\sim\exists z\, Sum\big(s(0), s\big(s(0)\big), z\big) \Leftrightarrow \forall z \sim Sum\big(s(0), s\big(s(0)\big), z\big) \Leftrightarrow Sum\big(s(0), s\big(s(0)\big), z\big) \rightarrow FALSE$
[23] This is a clause with all the literals negated, that is $\{\sim L_1 \sim, L_2, \dots, \sim L_n\}$.

that is the *subgoal* L_k is removed from the *goal* and is replaced by the tail of the clause of the program that unified with that *subgoal*.

A *SLD-refutation* is found as soon as we solve all remaining *subgoals*. When this happens the empty clause is derived and execution of the program ends.

Figure 15 shows an SLD-refutation for the set of clauses $S = \{C_1, C_2, C_3\}$, where C_3 is the query. On the figure, we emphasize the linear structure of the derivation (made clear using color) and the fact that the selection function has no importance on this example, since all goals have only one literal.

The execution of a program returns a substitution θ such that $\mathrm{P} \vDash \forall(C\theta)$.[24] This substitution is obtained through the composition of all the substitutions used to unify the goal and each of the clauses of the program. If an SLD-refutation is achieved after *n* applications of the resolution rule, then $\theta = \theta_1\theta_2 \ldots \theta_n$. The *answer* of the program is θ restricted to the variables that appear on the initial query. On the example of Figure 15 the substitution θ is

$$\theta = \theta_1\theta_2\theta_3 = \left\{ {}^{x_1}/_{s(0)}, {}^{y_1}/_{s(0)}, {}^{z_1}/_{z}, {}^{x_2}/_{s(s(0))}, {}^{y_2}/_{0}, {}^{z_2}/_{z}, {}^{x_3}/_{s(s(s(0)))}, {}^{z}/_{s(s(s(0)))} \right\}$$

and the *answer* of the program to the query $\leftarrow Sum(s(0), s(s(0)), z)$ is $\theta_a = {}^{z}/_{s(s(s(0)))}$, which fortunately means number 3!

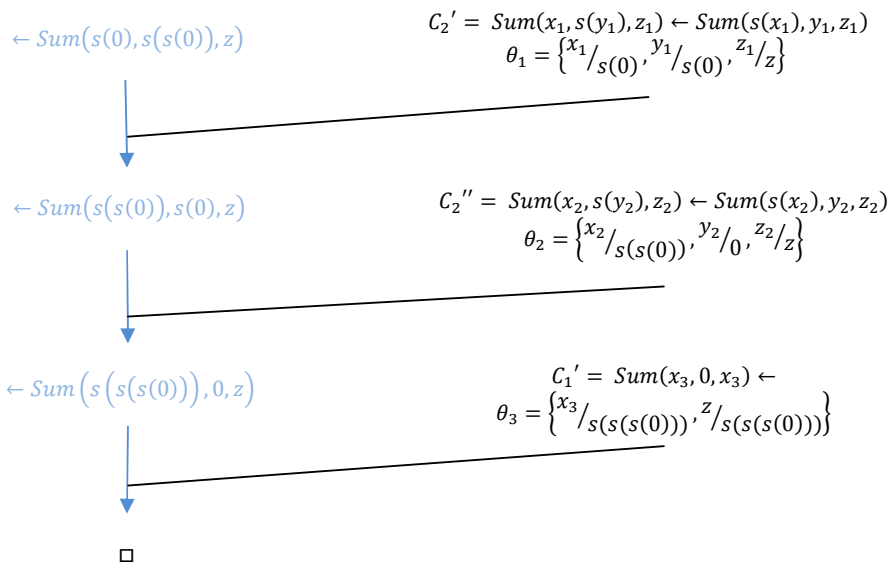


Figure 15 – SLD-refutation of program P_1 with the query $C_3 = \leftarrow Sum(s(0), s(s(0)), z)$.

PROLOG: executing a logical specification

Turning logical clauses into executable procedures permits that the logical specification of a computer program is itself its implementation. Nevertheless this change of perspective shifts our attention from theoretical

[24] Previously we asserted that "the execution of a logical program P can be seen as a derivation of a clause C, such that $\mathrm{P} \vDash \exists$C". Please note that $\mathrm{P} \vDash \forall(C\theta)$ implies the previous statement.

concerns to implementational efficiency. Unless we have an efficient way of implementing SLD-resolution, logical programming will have no practical use.

PROLOG implements an efficient *backtrack search* on the space of *SLD-refutations*.

The *selection function* chosen is $\psi(G) = \sim L_1$ where $G = \leftarrow L_1, L_2, \ldots, L_n$, that is given a goal clause G the selection function will always choose the left-most subgoal. This implies that an ordering relation is established among literals on goals.

We also need to establish some criteria on the selection of clauses of the program (that from now on will be considered as an ordered list), whenever a *subgoal* unifies with more that one of those clauses. We will assume the *standard ordering rule*: the first clause of the list that unifies with the goal elected by the selection function will be the chosen one.

An example of the search tree generated by PROLOG for the execution of a program is presented on Figure 16. We will assume from now on, and whenever we use PROLOG as an example, the usual PROLOG convention for variables, constants and predicate symbols: variables (e.g. X, Y) are presented in upper case and constants (e.g. a, b, c) and predicate symbols (e.g. p, q, r, s) in lower case.

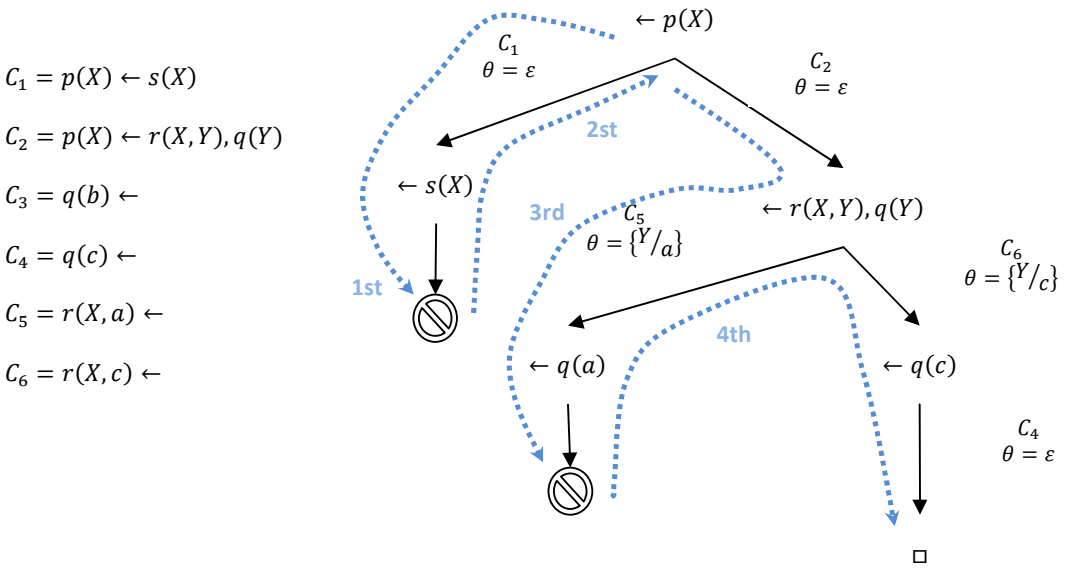


Figure 16 – Search tree generated by the execution of the program at the left, with the query $\leftarrow p(X)$.

On the *SLD-refutation search tree* above we emphasize the following facts:

- Each node of the tree contains the goal clause that still remains to be solved;
- Each arrow contains the clause of the program that unified with the parent and each respective unification;
- The subgoal chosen on the node $\leftarrow r(X,Y), q(Y)$ was $r(X,Y)$, since we are assuming the *selection function* described before;

- The symbol ⊘ identifies the absence of a valid unification between the literal chosen by the *selection function* and the head of any clause of the program. Whenever that happens, the search backtracks to the parent goal, and searches for alternative unifications for the literal.
- The *answer* to the initial query $\leftarrow p(X)$ is $\theta = \varepsilon$.
- The dotted arrows show the sequence of exploration of the tree and make the standard ordering rule clear: $\leftarrow p(X)$ unifies first with C_1 and after backtracking it unifies with C_2.

After the example it is obvious that a *depth-first search* is performed (rather than a breath-first search as shown on section 3.1.5). The *backtracking* property of the search implies that, rather than generating all the successors of a node at once, only one successor is generated at a time. In case of failure on a branch, then additional resolution operations are performed with other clauses of the program. On the previous example, resolution with clause C_6 is only performed after reaching a failure on the branch of clause C_5.

4. A brief comparison

After studying *SNIP* and *resolution*, we will now try to establish a comparison between both. On section 2.3.1 and while we were describing the types of inference available on SNePS, we restricted our domain of study to *backward node inference*. On this section, we could compare this inference mechanism with the general *resolution* mechanism. However this would make our comparison broad and consequently vague. Therefore we prefer to restrict our analysis to *SLD-resolution*, as this refinement is indistinguishable from PROLOG and it resembles backward inference.

4.1. Different aims, different concerns

The first thing that is worth understanding is that it makes no sense to claim one system as better than the other. Any comparison of this type is not fair, since the purpose of each of the mechanisms we are studying is different. While SLD-resolution was devised exclusively with the intention of performing automated theorem proving such that logical programs could be executed, SNIP is a software package that performs inference on the top of a knowledge representation formalism mainly driven by concerns founded on expressiveness and natural-language processing (see (Shapiro, 1999)).

The implications of this fact can be seen at the level of the information structures manipulated and at the level of the complexity of the inference rules used.

4.1.1. Difference of Expressiveness and complexity of data structures

On this section we will consider the expressiveness of a formalism, as the easiness to represent knowledge using it.

In PROLOG, only Horn clauses are allowed. Although the expressiveness of these clauses is limited, they can be represented using basic data structures: we can simply use a list of pairs antecedents/consequent to represent a program and another list to represent the goal to be solved.

On the other hand the expressiveness of our semantic network is much higher, since SNePS is a higher order formalism and a series of case-frames are available to express different logical relations between clauses. Nevertheless the data structures needed are also more complex since a directed acyclic graph is necessary.

To make this difference of expressiveness clear, we can look to Table 1 and convince ourselves that a single and-entailment is more powerful than a Horn clause. How many Horn clauses are necessary to represent a single and-or node?

4.1.2. Complexity of inference rules and of the deduction process

The difference in expressiveness is related to the number and complexity of each inference rule available and consequently to the complexity of the process that implements them.

PROLOG contains one single inference rule, [25] and the SLD-resolution inference process can be easily implemented using a backtracking search and a simple unification algorithm.

SNIP needs to implement at least five inference rules, one for each case-frame, and the inference process is dependent on the network-like structural organization of knowledge since deduction is an activation of the network itself.

4.1.3. Machine-like vs. Human-like Reasoning

Resolution is not concerned with the quality of the proof it finds, in the sense that it can or not be followed by a human, or presents a human-like "way of thinking". The simplicity of the resolution rule, although appropriate for a machine, produces proofs that are too long and where each step presents a very small logical inference step. Resolution is concerned with showing logical entailment rather than presenting a proof.

On the other hand SNePS is concerned with retrieving a human readable proof. The inference system implemented by SNIP is closer to a natural deduction system than resolution, and as a result produces a human-like reasoning type.

4.2. Comparing the inference process

4.2.1. An example

In this section we will use the example illustrated on Figure 17 and Figure 18 to compare relevant features of the inference process of both systems. We will assume that we have the PROLOG implementation of the program at the left of Figure 17, and that we also have the semantic network representing the same set of clause (Figure 18 presents the representation of only one of the clauses). Our aim is to derive clause $s(c)$ using both mechanisms.

The tree at the right of Figure 17 shows the search tree for the program at the left, with initial query $\leftarrow s(c)$. The left-side branch of the tree unifies the subgoal chosen by the selection function, $p(X,Y)$, to clause C_2. However this branch fails after solving the "very difficult" subgoal $q(a,c)$, and backtracking occurs. Then $p(X,Y)$ is unified again this time to the next clause of the program C_3. The tough subgoal $q(a,c)$ appears once again and needs to be solved one more time. Finally a refutation is found.

[25] Factorization is not necessary to assure refutation completeness on SLD-refutation.

$C_1 = s(Z) \leftarrow p(X,Y), q(Y,Z), r(X)$

$C_2 = p(d,a) \leftarrow$

$C_3 = p(b,a) \leftarrow$

$C_4 = q(a,c) \leftarrow \begin{pmatrix} very\ complex \\ deduction! \end{pmatrix}$

$C_5 = q(b,c) \leftarrow \begin{pmatrix} very\ complex \\ deduction! \end{pmatrix}$

$C_6 = r(b) \leftarrow$

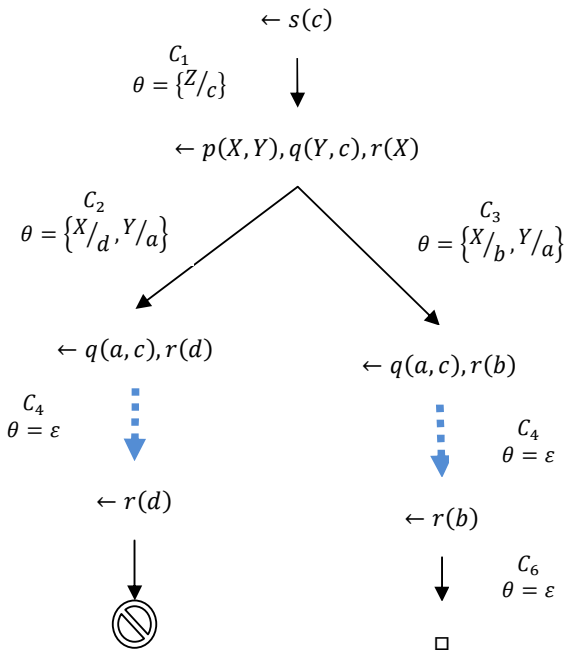


Figure 17 – Search tree for query $\leftarrow s(c)$ and the program at left. The tree underlines the redundant deduction of the subgoal $\leftarrow q(a,c)$ due to backtracking (the dotted arrow shows the deduction of this subgoal).

Let's now see how this problem is handled by SNIP:

- A temporary node is created to represent the literal $s(c)$. This node matches the pattern node P4 with binding $\{V3/c\}$;
- P4 sends a request through a channel parallel to the arc *cq* to the node M1!;
- Since M1! is asserted, it handles the request from P4 sending independent requests to each antecedent. These requests bind the variable V3 to the constant c;
- After receiving the request from M1!, node P1 tries to solve $p(X,Y)$, matching against other nodes on the network. Two solutions are obtained, $\{V1/d, V2/a\}$ and $\{V1/b, V2/a\}$, which are reported to M1!. Consequently M1! is scheduled on the high priority queue by P1 but no consequents can be inferred.
- P2 handles the request from M1! in a similar way. Two other solutions are found: $\{V1/a, V3/c\}$ and $\{V1/b, V3/c\}$ (remember that the request from M1! binds V3 to c, and no solutions where V3 is bounded to other variable are retrieved). M1! Is informed but no consequents can be inferred again
- Finally the antecedent P3 finds solution $\{V1/b\}$ and reports to M1!;
- M1! can now produce a consequent. From the RUI

  $$\left(\{V1/b, V2/a, V3/c\}, 3{,}0, \{P1: true, P2: true, P3: true\}\right),$$

  the consequent $s(c)$ is inferred.

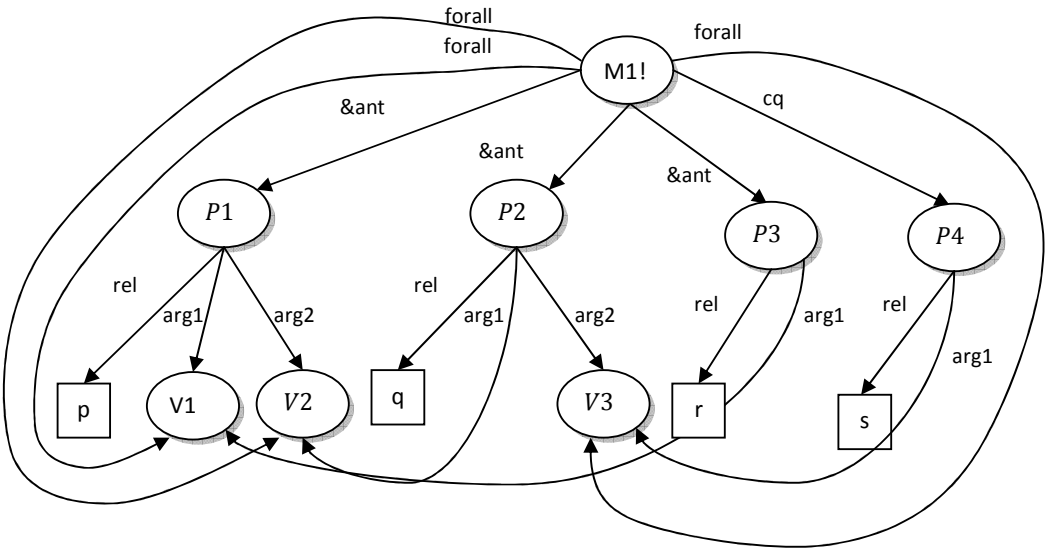


Figure 18 – SNePS rule node representing clause $s(Z) \leftarrow p(X,Y), q(Y,Z), r(X)$.

4.2.2. Comparing search strategies

PROLOG implements a backtracking search on the SLD-refutation space. This search procedure has its origin on three conventions:

- the selection function chooses the left-most literal of the goal;
- the antecedents of a rule are added before all other subgoals, in the order they appear in the clause. This property together with the previous one induce a depth-first search;
- only one branch of the tree is generated at a time, that is, when the left-most literal is chosen, only one unification is performed between this literal and the head of the clause that appears first on the list of rules of the program. Additional unifications are made only in case of failure on the branch.

This strategy is completely different from the one used by SNIP. Firstly because in SNIP, a breath first search is made rather than a depth-first search. Whenever a process is activated it is scheduled at the end of the MULTI queue, not at the beginning. On the other hand PROLOG adds new subgoals to the beginning of the goal, not to the end, and assumes the selection function previously described.

Next, whenever a set of processes attached to the antecedents of a rule node are scheduled in the end of the queue, there is no ordering relation between them. On PROLOG they are inserted by the order they appear in the clause.

Finally the *match algorithm* performs unification with the entire network. Therefore all *match channels* are established at once and all the processes associated to the matched nodes are scheduled on the queue in the same instant.

4.2.3. Redundant deductions

In the subsection "Producer-consumer model" of section 2.3.4, we described each SNePS process associated to a node of the semantic network as a *data collector*. One of the main consequences of this property was the fact that no redundant deductions were made: each producer maintains a list of known-instances of the node, and reports them in case of being asked by a consumer.

The SLD-resolution mechanism is not protected against redundant deductions. Our example shows how these can happen in case of backtracking, but exactly the same subgoal can be produced twice if two different rules have the same literal as an antecedent.

The previous example clearly shows redundant deductions. While SLD-resolution solved subgoal $q(a,c)$ twice, SNePS solved it only once. Additionally, SNePS also produced a solution to the also "very complex" subgoal $q(b,c)$. Both solutions were stored on the KNOWN-INSTANCES register of the process attached to P2.

4.2.4. Binding of common variables

We now analyze the way variables on antecedents are instantiated during the deduction process, in particular when these variables are shared among more than one antecedent. While the antecedents of a Horn clause are always a conjunction, on SNePS this only happens when we have an and-entailment rule node. This is the case we want to focus now.

The example given in the beginning will be used again and we will start by analyzing PROLOG. When clause $\leftarrow p(X,Y), q(Y,c), r(X)$ is resolved with C_2, the *resolvent* obtained is $\leftarrow q(a,c), r(d)$. On this clause, Y appears bound to a, as the selected subgoal $p(X,Y)$ unified with the head of C_2, $p(d,a)$. The search for an *SLD-refutation* continues on this branch, trying to solve this time $q(a,c)$, the next left-most literal chosen by our *selection function*.

We will now turn to SNePS. When M1! sends requests to each antecedent (P1, P2 and P3), those requests are sent simultaneously and in parallel. Each process works independently and variable bindings found on one antecedent are not communicated to other antecedents. On our example, although V2 is shared by pattern nodes P1 and P2, when P1 finds solution $\left\{{V1}/{d}, {V2}/{a}\right\}$ this does not guide solutions on P2. Antecedent P2 will find all solutions with V2 unbound, will report them to rule-node M1!, and this last node will be in charge of finding a compatible instantiation for the variables nodes quantified universally with an arc **forall**. (Choi, et al., 1992) devises a method for controlling the combinatorial problem that arises on conjunctive nodes, combining *RUI*'s on a tree-like structure, and positioning literals with common variables adjacent on that tree.

Independent resolution of antecedents is desirable because it allows parallelization on multiprocessing or multithreading systems. On SNePS, node processes are independent of each other. However parallelization with total absence of communication leads to clear inefficiencies. We return to our example to clarify this aspect, assuming that clauses C_2 and C_3 are not present on our knowledge base.[26] If this is the case, there is no

[26] We assume that the corresponding network nodes are also not present.

solution for $p(X,Y)$. The absence of an SLD-resolution for query $\leftarrow s(c)$ is immediately detected after the second search step, when that literal is selected for resolution and fails to unify with any clause of the program. On SNePS, all the processes attached to antecedent nodes are activated and failure of P1 is not propagated to brother nodes.

4.2.5. Recursive Rules

According to (McKay, et al., 1981), a recursive rule is a rule of the type $A_1 \wedge \ldots \wedge A_n \rightarrow B_1 \wedge \ldots \wedge B_m \rightarrow C$ where C unifies with at least one of the antecedents of the sequence. This type of rules generally represents a problem, since it may cause theorem provers to enter a cycle. Circular definitions and equivalences are all recursive rules and suffer from this problem. The classical example of a recursive rule, also discussed in the previously cited paper, is the definition of the family relation "ancestor", presented on Figure 19.

$$C_1 = ancestor(b,c) \leftarrow$$

$$C_2 = ancestor(a,b) \leftarrow$$

$$C_3 = ancestor(X,Z) \leftarrow ancestor(X,Y), ancestor(Y,Z)$$

Figure 19 – Definition of the relation "ancestor" and the relationship between three members of a family: *a* can be the parent of *b* and *b* can be the grandfather of *c*.

We will use this example to show the way both systems handle this type of rules.

PROLOG tries to solve the problem assuming the *standard ordering rule*. If we consider that when a subgoal unifies with the head of more than one clause, the one that comes first in the list is the one chosen first, then a careful ordering of clauses of the program can avoid entering an infinite loop. For this reason, generally unit clauses of the form $C \leftarrow$, which are called facts, come first on the list. Similarly clauses which have heads more specific than others also appear before these.

Figure 20 shows the search tree for the program of Figure 19. The order of the solutions found by the PROLOG is top to bottom, left to right.[27] If instead of the ordering presented before we had chosen P = (C_2, C_1, C_3), then C_2 would be selected first on the left-most branch of the second level and we would avoid the infinite branch. If this would be the case, we would also have found the solution $ancestor(a,c)$ before looping forever.

[27] Most PROLOG implementations halt after finding a refutation, that is after finding a branch ended by a □. However it is generally possible to request for alternative solutions to the interpreter. When this functionality is available, PROLOG continues backtracking and searching for alternative refutations. Our SLD-search trees assume a top to bottom, left to right order on the finding of refutations.

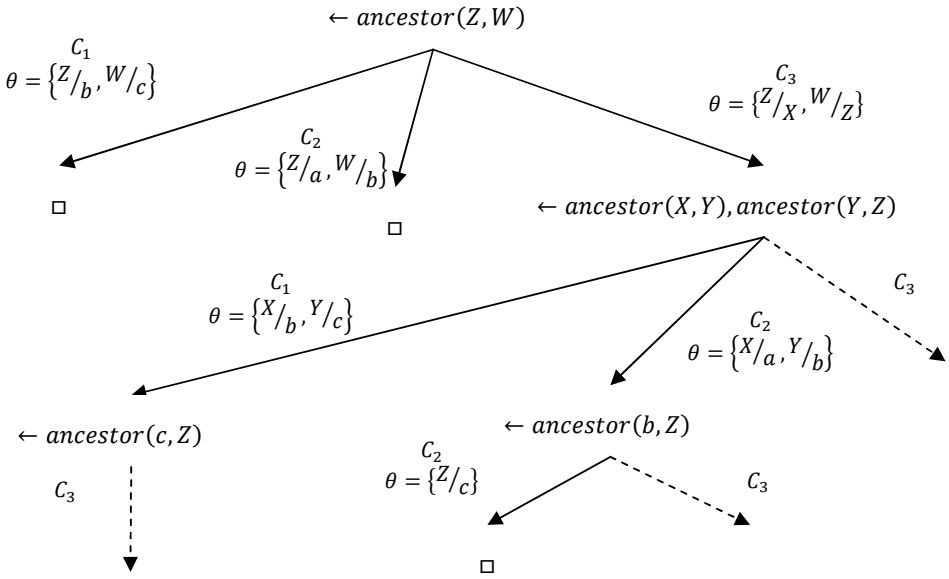


Figure 20 – Search tree for the program of Figure 19.

It is obvious that irrespectively of the order of the previous three clauses, the search tree will always have infinite branches. To try to avoid that we could redefine the $ancestor$ relation as Figure 21 suggests.

$$C_5 = ancestor(X, Y) \leftarrow parent(X, Y)$$

$$C_6 = ancestor(X, Z) \leftarrow parent(X, Y), ancestor(Y, Z)$$

Figure 21 – Alternative definition for the predicate $ancestor$.

This time instead of giving facts using the relation $ancestor$ we could use the relation $parent$: $parent(a, b)$ and $parent(b, c)$. However even using this new definition, changing the order of the literals on the antecedent of the clause C_6 also gives rise to infinite branches during the backtrack search. [28]

After this the morale is simple: to effectively code recursive rules on PROLOG, the logic programmer must be perfectly conscious of the way backtracking search mechanism works that is, he must keep the PROLOG search strategy in his mind (its selection function and the *standard ordering rule*).

We finish our discussion of PROLOG with two relevant notes:

- First that the problem of failing to find some refutations only arises because the search strategy used by PROLOG is not fair. If we explore a search tree with a breath first strategy it is easy to see that, although the process may never terminate, all refutations would be found after a finite amount of

[28] If we use the rule $C'_6 = ancestor(X, Z) \leftarrow ancestor(Y, Z), parent(X, Y)$ and considering that the selection function always chooses the left-most literal of the goal, then the chosen literal $ancestor(Y, Z)$ would possibly unify first with C_5 and then with C'_6. An infinite branch on the search tree would always emerge from this last unification. Using C_6 rather than C'_6, ends the branch as soon as there is no unification for $parent(X, Y)$, avoiding infinite branches on the tree.

time. The completeness of resolution is not at stake; we simply fail to find refutations due to the non-fair strategy.

- Next that the notion of semantic consequence $\Gamma \vDash \varphi$ that we wanted to capture with PROLOG is somehow changed. The fact that we cannot look to Γ as a set but rather as an ordered tuple, is a clear evidence of this. In fact as (Cruz-Filipe, 2006) refers, two formulas may be logically equivalent but not operationally equivalent. [29]

SNePS offers a more interesting approach to recursive rules which emerge, once again, from the fact that SNePS' processes are *data collectors* (see subsection "Producer-consumer model" on section 2.3.4). An immediate consequence of the fact that a node never produces the same solution twice to the same consumer is the evidence that the inference process never enters a loop.

Figure 22 represents the same information of Figure 19, this time using a semantic network.

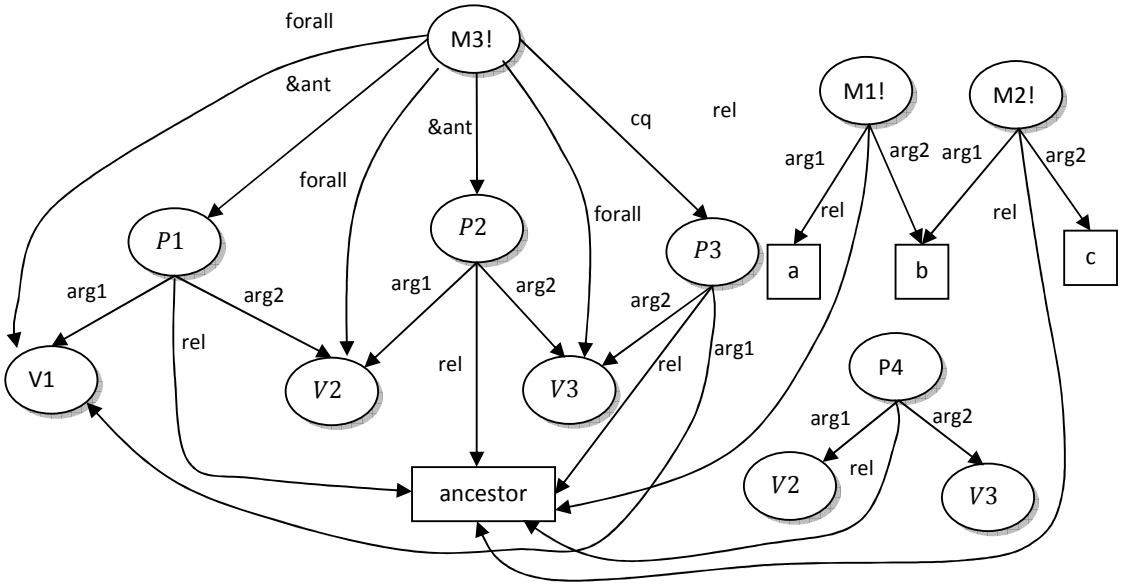


Figure 22 – SNePS semantic network for the clauses of Figure 19. The node P4 is a temporary node created when a deduction is asked to the system.

As we have done in previous sections, we will describe the guidelines of the deduction produced when the SNePS command **(deduce rel ancestor arg1 $who1 arg2 $who2)** is inserted. The entire deduction process is given as an Appendix, on section 9.1.

1. First the temporary node P4 is created and the match operation is invoked. P4 unifies with the antecedents of the rule node P1 and P2, with the consequent P3 and obviously with molecular asserted nodes M1! and M2!. A request is sent to all these nodes.
2. Nodes M1! and M2! are asserted and correspond to solutions. These solutions are reported to node P4.

[29] Just remember the discussion about clauses C_6 and C'_6.

3. The antecedent nodes P1 and P2 also receive a request. However there is no *cq* arc pointing to them and they cannot perform a match operation because the request comes from a match channel;[30]
4. Node P4 sends a request to the rule node M3!, asking for derivations of the consequent.
5. As M3! is asserted, it tries to prove instances of the consequent P4. For that purpose, it sends requests to the antecedents P1 and P2.
6. The antecedents P1 and P2 perform a match operation on the network. P1 unifies with P2, M1!, and M2!. P2 unifies with P1, M1!, M2!. It is very important to note that neither P1 nor P2 unify with the consequent of the rule M3!.
7. P1 finds two solutions: $\left\{{}^{V1}/_{a}, {}^{V2}/_{b}\right\}$ which corresponds to M1! and $\left\{{}^{V1}/_{b}, {}^{V2}/_{c}\right\}$ which is obtained from M2!. The same happens to P2: $\left\{{}^{V2}/_{a}, {}^{V3}/_{b}\right\}$ and $\left\{{}^{V2}/_{b}, {}^{V3}/_{c}\right\}$.
8. Both P1 and P2 return reports to the rule node M3!, containing the solutions found. M3! produces the solution $\{{}^{V1}/_{a}, {}^{V3}/_{c}\}$, that is reported to P4 through the consequent P3.

Given that the antecedents of M3! do not unify with the consequent of the same node, a channel between P1/P3 and P2/P3 is not established. Therefore any solutions derived by the rule are not reiterated to the antecedents of the same rule. This fact has a rather disappointing consequence: the SNePS deduction system is incomplete. As an example if $ancestor(c, d)$ would be inserted in the network, $ancestor(a, d)$ would not be derived with the SNePS command given before.

A way to overcome this "technical problem" related to the non unification described is to use **(deduce rel ancestor arg1 a arg2 $who2)**.[31] In this case $ancestor(a, b)$, $ancestor(a, c)$ and $ancestor(a, d)$ would all be derived, given that the unification of P1 and P3 would occur.

After this, the idea to retain is simple: although incomplete, SNIP does not suffer from the looping problem caused by recursive rules since every node implements the data-collectors paradigm and no duplicate answers are produced.

[30] Remember that matching again would result into an infinite sequence of matches. In fact if *n1* performs a match operation and finds *n2*, if *n2* would match again the operation would be redundant since it would find the same nodes.

[31] With this SNePS command, when P3 sends requests to the antecedents P1 and P2, it sends with the initial binding $\{{}^{V1}/_{a}\}$. As a consequence, all the variables shared between P1 and P3 are instantiated and therefore P1 will unify with P3.

5. Controlling antecedents

On section 4.2.4, "Binding of common variables", we discussed how SNePS' approach to conjunctive goals differed from PROLOG's approach. While on the first case, antecedents are potentially run in parallel and the failure on one of them does not influence the others, PROLOG tries to solve antecedents sequentially, following the order they appear on the rule, and the results obtained for one of the antecedents conditions the execution of the others. As we pointed before, on PROLOG the influence among antecedents goes beyond the decision of running or not the next antecedent, as variable binding on the antecedents that follow depends on the instantiation found for the previous antecedents.

If we shift our attention to the efficiency of the deduction process then, from this analysis, it is arguable that SNePS' approach can lead to clear inefficiencies on the use of computational resources. In this section we describe the problem we want to tackle, propose solutions, choose and describe the implementation of one of them and finally show some experimental examples.

5.1. Running useless antecedents: the problem

Let's admit that node $Mn!$ is an and-entailment rule node (Figure 23).[32] As we have shown, when a consequent C_i, $1 \leq k \leq m$, needs to be proved, $Mn!$ sends requests in parallel to all the antecedents A_i, $1 \leq k \leq n$. As a result all antecedents are scheduled on the low priority MULTI queue, say by the order of their numerical index. [33]

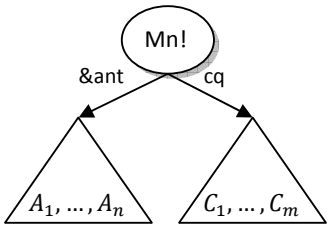


Figure 23 – And-entailment rule node.

Let's suppose now that antecedent A_k, $1 \leq k \leq n$, has no solution on the network. As a consequence A_k will never report to the rule node $Mn!$. Actually we can go a little further and assume that A_k has solutions but none of them is compatible with the solutions of the antecedents that already reported (assuming that they share some variables).

In these cases it's safe to admit that the and-entailment rule will not produce any instances of the consequents, since a solution to all the antecedents is needed to apply the inference rule. Nevertheless given that requests

[32] Latter on, we will extend this reasoning to other types of rule nodes.
[33] Remember that this order is not defined on SNePS as we referred on section 4.2.2 - Comparing search strategies. Although antecedents are scheduled in a deterministic way, the order is neither controlled by the user nor described on the SNePS' manual.

are sent in parallel and the antecedents are scheduled on the MULTI queue in the same instant, the antecedents that still remain to be executed upon failure of the deduction of one of the antecedents will be unnecessarily run. This is undesirable if we are concerned with the efficiency of the system.

To sum up, on this chapter we derive conditions that, based on the failure to find solutions to some antecedents of the rule nodes (not necessarily and-entailment nodes), allow us to avoid the execution of some antecedents without affecting the number of solution found. This will spare resources and produce eventually faster inferences.

5.2. How antecedents can be idled or have their execution stopped

We found two possible solutions to the problem. On this section we sketch them briefly, using the and-entailment rule node as an example, but keeping in mind that we could adapt similar ideas to other types of rule nodes.

5.2.1. Implementing the STOP and DONE messages

The SNePS' technical report (Hull, 1986) suggests the implementation of two messages, STOP and DONE, that could be used to control the deduction process of SNePS. Although not present on the official version of SNePS, (Mamede, 1992) works on this topic on his version $SNePS_R$.

Basically a node would send the DONE message when "it has already sent out all of the instances of itself that it possibly can" and would send a STOP message to halt the execution of other processes. These messages could be used to produce the desired objective: all the antecedent would continue to be scheduled in the MULTI queue in the same instant, would freely execute themselves and if an antecedent A_k would return the message DONE to an and-entailment rule node without previously producing any solution (or a compatible one), then the rule node would send a STOP message to all the other working "brother" antecedents.

This solution is rather elegant because it uses the communication channels installed between processes but has some inconvenients:

- it increases the complexity of the communication protocol and therefore the complexity of the behavior of the processes;
- it doesn't completely prevent the execution of useless processes: all processes are scheduled on the MULTI queue, may run and eventually may have their execution stopped if they receive the STOP message;
- the DONE and STOP messages need to be passed through the whole chain of producers and consumers. For instance if some antecedent receives a STOP and had previously requested other nodes, this antecedent needs to propagate the STOP message to all the producers that were already requested in the past and had the antecedent as their boss.

Because of these drawbacks we decided to try to find other solutions.

5.2.2. Changing MULTI: creating an idle queue

Another option that deserved more attention was the creation of another queue on the MULTI system that would keep processes which execution would be held. The emergence of reports from working antecedent processes could take idle "brothers" from that state and reschedule them on the low priority queue. If none of the working processes would report, then idle processes won't be executed.[34] The condition that allows an antecedent process to be idled is simple: if none of the working antecedents that share the same rule node report results, then it is irrelevant whether the idle antecedents report or not, as the corresponding rule node will not have enough information to be applied anyway.

We illustrate this with the and-entailment example of Figure 23: if we schedule $n-1$ of the n antecedents on the idle queue and allow only one of the antecedents to be executed, if this working antecedent doesn't produce reports then, for sure, the $Mn!$ rule node will not derive new consequents. Therefore the $n-1$ idle antecedents can be "forgotten" on the idle queue.

In order to implement this method, we can simply inspect the requests and reports of the processes that are either scheduled or picked to execution by MULTI:

- when a rule node schedules an antecedent on the low priority queue,[35] MULTI checks the REQUESTS register. If a request from a rule node is detected, then MULTI checks the type of rule node that signs the request and the state of its RUI structures (which contain the already derived instances for all the antecedents). Accordingly, MULTI can schedule the process in the low priority queue or keep the process in the idle queue, waiting for reports of the working processes.

  On our example of Figure 23, after being asked by $M_n!$ to schedule all A_k, $1 \leq k \leq n$, on the low priority queue, MULTI checks the REQUESTS register of the initiated processes and finds requests from an and-entailment node. As none of the other antecedents returned any instance yet (information which can be obtained from the state of the consequent channels of the rule node), MULTI decides to schedule one of them on the low priority queue and idle the remaining. The selection of which nodes remain in the idle queue and which are selected to work will be discussed latter on the subsection "The order of the antecedents matters!".

  Figure 24 details this interaction. Rule node $M_n!$ decides to initiate antecedent A_k. In order to decide whether to choose the LOW or IDLE queues, MULTI checks RUI structures of the consequent channels of the rule node. Finally, MULTI decides to schedule the process on the IDLE queue.

[34] ... or could eventually be executed after all the processes of the low priority queue.
[35] This is achieved calling the MULTI function **initiate.**

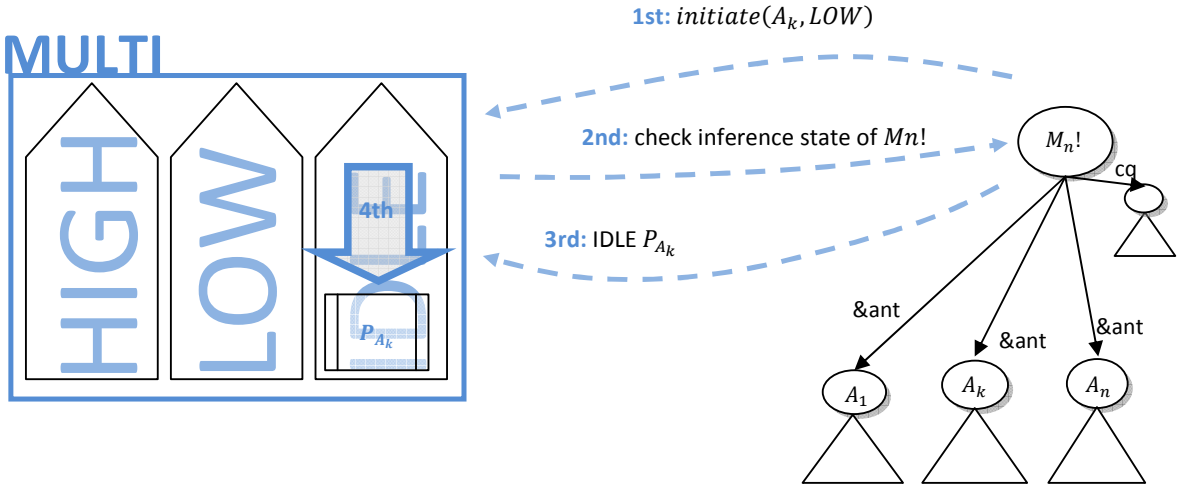


Figure 24 – Interaction between MULTI and the rule node M_n!, when that node decides to initiate the antecedent A_k.

- when the working antecedents report new solutions, MULTI checks again the state of the consequent channels of the rule node which receives the report, and concludes whether it is appropriate to take any antecedent from the idle state. Again, by now, we aren't concerned with the selection of the idle antecedent.

  Using our example, if one of the antecedents of the and-entailment reports, then it may be appropriate to take another idle antecedent from that state, as Figure 25 suggests. In this figure the antecedent A_j reported a solution to the rule node M_n! and as a consequence the process P_{A_j} was scheduled on the high priority queue. After the execution of this process, MULTI needs to check the inference state of the rule node. After checking RUI structures, P_{A_k} was rescheduled on the LOW priority queue.

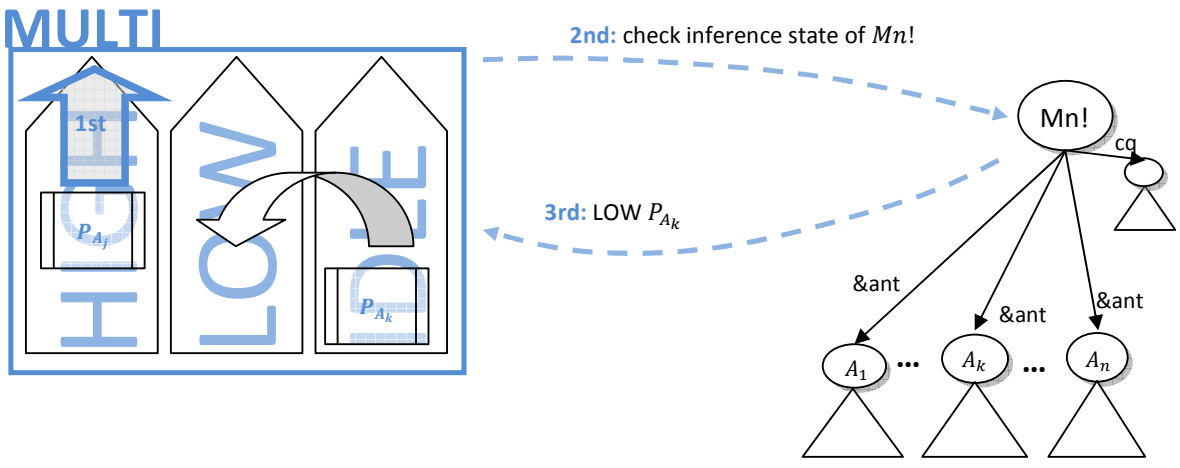


Figure 25 – A process scheduled on the IDLE queue is rescheduled on the LOW priority queue due to a report from an antecedent.

This solution solves some of the problems that arose with the previous approach:

- although initiated on MULTI, idle processes may never be executed at all: if no reports are produced, idle processes will remain in the IDLE queue.
- no increase on the complexity of the behavior of nodes;
- implementation can be accomplished through minor changes on MULTI and an intermediate layer of code that checks REQUESTS and REPORTS registers and takes appropriate measures according to them.

These advantages made us choose this second approach to the problem.

5.3. Finding conditions to schedule antecedents on the idle queue.

Until now we have been using the easily understandable and-entailment example to illustrate how some antecedents can be idled without affecting the results of the inference process. However this concept can be extended to other types of rule nodes. In this section we show how the type and inference state of a rule node affect the state of the antecedents of that node. Therefore we want to understand which nodes should be executed and idled, such that we don't lose possible inference solutions.

5.3.1. Or-entailment, And-entailment and Numerical-Entailment

We will investigate numerical-entailment first, as and-entailment and or-entailment are particular types of the former.[36] The inference rule associated to

$$\{A_1, \dots, A_n\} i \Rightarrow \{C_1, \dots, C_m\},$$

in order to be applied, must find i true antecedents. Thus, if we execute $n - i + 1$ antecedents and none of them reports any solution, the remaining $i - 1$ idle processes will not be enough to activate the rule. Therefore we can leave these last antecedents in the idle queue without executing them.

Let's now admit that some of the working antecedents report solutions, that the consequent channels of the rule node are updated and that the RUI with the highest number of true inferences is generically given by

$$\left(\sigma, pos_{i\Rightarrow}, negs_{i\Rightarrow}, \{A_1 : true, A_2 : true, \dots, A_{pos} : true, \dots\}\right).$$[37]

In this case it remains to derive $i - pos_{i\Rightarrow}$ true antecedents to activate the inference rule. As a result the number of working antecedents should be switched to

$$working\_ants_{i\Rightarrow} = min\{n, n - i + pos_{i\Rightarrow} + 1\}.$$

Each time the inference state is checked on Figure 24 and Figure 25, $working\_ants_{i\Rightarrow}$ is recalculated and that number of antecedents is rescheduled on the LOW priority queue. For the moment we won't be concerned with which antecedent we should pick. We will just assume that antecedent have a given order that doesn't

[36] V $\Rightarrow$ is equivalent to $1 \Rightarrow$ and &$\Rightarrow$ is equivalent to $n \Rightarrow$ where n is the number of antecedents.
[37] We suggest the reader to briefly reread sub-section "Production of solutions on rule-nodes".

change and that we always pick the first $working\_ants_{i\Rightarrow}$ antecedents following that order (the subsection that follows "The order of the antecedents matters!" deals with that problem).

We also note that RUI tuples assure compatibility among variable instantiations of antecedents. Thus if two antecedents report with incompatible solutions, $pos_{i\Rightarrow}$ will be increased at most by 1. This is clearly the desired behavior.

Focusing now on and-entailment and or-entailment, the previous formula shows what we are expecting. And-entailment will have one single antecedent without true solutions running at a time

$$working\_ants_{\&\Rightarrow} = min\{n, pos_{\&\Rightarrow} + 1\},$$

and or-entailment won't idle any of its antecedents and at each moment the number of working antecedents will be

$$working\_ants_{\vee\Rightarrow} = min\{n, n + pos_{\vee\Rightarrow}\} = n.$$

5.3.2. And-or

The previous analysis can be adapted to and-or rule nodes of the form

$$\wedge\vee_{min}^{max}\{P_1, \dots, P_n\}.$$

However the previous discussion differs in at least three points (remember Table 1):

- and-or produces inferences based on both true and false antecedent solutions;
- there are two inference rules at stake: one infers $n - max$[38] false consequents from max true antecedents (we will call it Rule 1) and the other min true consequents from $n - min$ false antecedents (from now on Rule 2);
- and-or is capable of deriving contradictions and we don't want our approach to affect that property.

Let $pos_{\wedge\vee}$ and $negs_{\wedge\vee}$ be the highest number of true and false solutions found for every RUI tuple in the consequent channels of the and-or rule node. Let num_ants be the number of antecedent of that consequent channel, that is $num\_ants = n - 1$.[39] Then the previously derived formula can be applied to both Rule 1 and 2. For the first case, the number of working antecedents should be

$$working\_ants_{Rule1} = min\{num\_ants, num\_ants - max + pos_{\wedge\vee} + 1\}.$$

For the second case, we will have

$$working\_ants_{Rule2} = min\{num\_ants, num\_ants - (n - min) + negs_{\wedge\vee} + 1\}$$

$$= min\{num\_ants, min + negs_{\wedge\vee}\}.$$

Finally the number of allowed working antecedents will obviously be the maximum of both values,

[38] We will call max and min to the numbers pointed by each respective arc. n is the number of arcs with label "*arg*"

[39] When an and-or or thresh node receives a request through a channel parallel to an *arg* arc, a consequent channel is established and all the remaining arcs are considered as antecedents. Therefore the number of antecedents that are sent requests is $n - 1$.

$$working\_ants_{\wedge\vee} = max\{working\_ants_{Rule1}, working\_ants_{Rule2}\},$$

as we don't want to prevent any possible inference to be made.

Considering now that in order to derive contradictions using and-or, we need more than max true or more than $n - min$ false antecedents, knowing that at each instant of the inference process we set the number of idle antecedents such that neither Rule 1 nor Rule 2 can be applied in case of failure of all working nodes and finally remembering that in order to apply those inference rules we need max true or $n - min$ false antecedents, then we can conclude that the derivation of contradictions won't be affected by the idleness of some antecedents.

5.3.3. Thresh

The differences between thresh and and-or nodes are minimal. Similarly, as Table 1 presents, thresh node implements two inference rules. Both rules have true and false antecedents and can only be applied, either to derive new instances of the consequents or a contradiction, when more than $n - threshmax + thresh - 1$ (true or false) antecedents are derived.

Let's admit that we found the RUI tuple with the highest sum of derived true and false antecedents and that we will represent that sum by $pos_\Theta + neg_\Theta$.

Using all this, we can define the number of working thresh antecedent as

$$working\_ants_\Theta = min\{num\_ants, num\_ants - (n - threshmax + thresh - 1) + pos_\Theta + negs_\Theta + 1\} =$$

$$= min\{num\_ants, threshmax - thresh + 1 + pos_\Theta + negs_\Theta\}.$$

Finally, knowing that contradiction can only be derived when the number of derived antecedents is higher that $n - threshmax + thresh - 1$, we can conclude that idling antecedents won't have influence on the derivation of contradictions, following a reasoning similar to the previous node.

5.3.4. Summarizing Results

Table 3 presents, for each node type, the number of antecedents that should work at each moment, according to the number or true and false derivations already performed by the node. The functions $pos(x)$ and $negs(x)$, where x is a RUI, return the number of true and false derivations, respectively.

Node Type	Chosen RUIs to calculate pos and $negs$	Number of working antecedents
&⇒	$pos_{\&\Rightarrow} = pos(RUI)$, where $RUI: \forall_{x \in RUISet}\ pos(x) \leq pos(RUI)$	$min\{n, pos_{\&\Rightarrow} + 1\}$
∨ ⇒	Not applicable	n
i ⇒	Same as for &⇒	$min\{n, n - i + pos_{i\Rightarrow} + 1\}$

Node Type	Chosen RUIs to calculate pos and $negs$	Number of working antecedents
$\wedge\vee_{min}^{max}$	$pos_{\wedge\vee} = pos(RUI_1), negs_{\wedge\vee} = negs(RUI_2)$, where $RUI_1: \forall_{x \in RUISet}\ pos(x) \leq pos(RUI_1)$ $RUI_2: \forall_{x \in RUISet} negs(x) \leq negs(RUI_2)$	$max\{r_1, r_2\}$, where $r_1 = min\{num\_ants, num\_ants - max + pos_{\wedge\vee} + 1\}$ $r_2 = min\{num\_ants, min + negs_{\wedge\vee}\}$
$\Theta_{\text{thresh}}^{\text{threshmax}}$	$pos_\Theta = pos(RUI), negs_\Theta = negs(RUI)$ where $RUI: \forall_{x \in RUISet}\ pos(x) + negs(x) \leq pos(RUI) + negs(RUI)$	$min\{num\_ants, threshmax - thresh + 1 + pos_\Theta + negs_\Theta\}$

Table 3 – Number or working antecedents for each rule node.

5.4. The implementation of the solution

This section does not intend to be an exhaustive description of the changes made to the code of SNePS to encompass the previous ideas. The following text simply focuses the most relevant details that were taken into consideration during the implementation of our solution.

5.4.1. Changing MULTI

The changes to the MULTI package where minimal and had the objective of executing the tasks described on section 5.2.2. Three changes should be enumerated:

- the creation of a new global variable called ***idle-queue***;
- the change of the **multip** procedure, which implements the MULTI cycle. This procedure picks a process from the highest non-empty queue and executes it. The purpose of the change was to assure that when reports are produced and high priority processes are executed, we take pending processes from the idle to the low priority queue if that is appropriate (Figure 25). For that purpose we need to check the state of the consequent channels after being updated with the new reports, and conclude whether the new RUIs created free any pending antecedent (function **update-state-due-to-report**) Figure 26 shows the pseudo-code of **multip** function.
- finally the change of the MULTI function **initiate**. This function, which is in charge of ordering the scheduling of a process on one of the queues, must be changed in order to also choose the idle queue. For that purpose the process invokes **determine-idle-or-low-queue**. This last function checks the requests that are present on the process, filters those that come from a rule node and in that case decides to schedule on the low or idle queue according to the inference state of the rule node. Figure 28 shows the pseudo-code of the **initiate** function. A problem that we didn't mention yet and that our function must handle is the possibility of two consumers requesting the same producer, not agreeing on its state. Figure 27 tries to picture that situation: node A_k was requested both by both rule node $Mn!$ and pattern node P_j (which matched A_k). When both bosses initiate P_{A_k}, **determine-idle-or-low-queue** checks the REQUESTS register, and concludes that while one of the senders suggests the

idle state, the other needs that process to work. In these cases, the process is obviously scheduled on the LOW priority queue.

```
function multip
global vars:   *high-priority-queue*,
               *low-priority-queue*,
               *idle-queue*,
               *final-idle-dequeue*, // true if idle processes are supposed to
                                     // be run when high and low priority queues are empty
local vars:   interesting-reports, // true if reports reach a rule-node and are sent by an antecedent node.
 {
        while ("there exists processes to run")  {
                interesting-reports ← false
                /*Choose highest priority non empty queue*/
                if (non-empty(*high-priority-queue*)) {
                        choosen-queue ← *high-priority-queue*;
                        interesting-reports ← decide-interesting-reports(first(*high-priority-queue*));
                }
                elseIf (non-empty(*low-priority-queue*)) {
                        choosen-queue ← *low-priority-queue*;
                }
                elseIf (non-empty(*idle-queue*) and *final-idle-dequeue*) {
                        choosen-queue ← *idle-queue*;
                }
        }
        /* Run First process of chosen queue */
        next-process ← first(choosen-queue)
        execute(next-process)
        /* Eventually take processes from idle state */
        if (*using-idle-queue* && interesting-reports) {
                update-state-due-to-report(next-process)
        }
}
```

Figure 26 – Pseudo-code of the multip function.

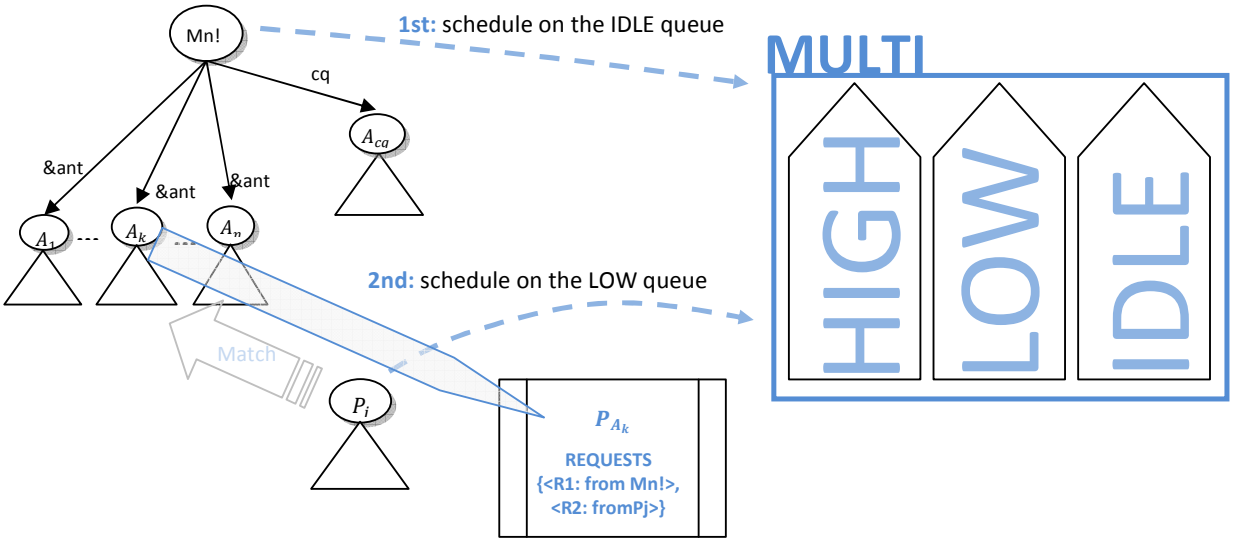


Figure 27 – Both $M_n!$ and P_j send requests to node A_k. However, while the first decides to idle A_k the second decides to set it to the working state.

```
function initiate (process)
global vars:    *high-priority-queue*,
                *low-priority-queue*,
                *idle-queue*.
local vars:     chosen-queue,  // queue chosen to store the process
                process-priority. // is set to the content of the process register *PRIORITY*
{
        process-priority ← regfetch(process, '*PRIORITY*)
        case(process-priority) {
                HIGH: chosen-queue ← *high-priority-queue*
                LOW: chosen-queue ← determine-idle-low-queue(process)
        }
        schedule(process, chosen-queue)
}
```

Figure 28 – Pseudo-code of the MULTI function initiate. The called procedure schedule inserts the process on the chosen queue, according to some policy.[40]

5.4.2. Iterating over all RUIs

We now switch to the problem of the calculation of the set of working antecedents. That set is maintained in a new register that was added to rule processes, ***WORKING-ANTECEDENTS***, which is initialized and updated on functions **determine-idle-or-low-queue** and **update-state-due-to-report**, respectively.

On section 5.3, we gave formulae for the number of working antecedents. That number was calculated based on the RUI data structures present on each consequent channel (to obtain pos and $negs$, the number of true and false inferences made by the rule node) and on the parameters of each node (check Table 3 again). Thus to find the RUIs that match the conditions on the middle row of Table 3, we need to iterate over all RUI structures.

While in previous versions of SNePS a simple list of RUIs was used, the work of (Choi, et al., 1992) introduced several different ways to efficiently represent RUI structures on nodes. The representation depends on the type and structure of the nodes. In this section, we describe those representations and explain how in one of the cases it can help us choosing the next appropriate antecedent to run.

Linear and S-indexing methods to store RUIs on non-conjunctive rule nodes

The linear method corresponds to the original method used on SNePS and stores RUIs on a simple set. The method is used for all the situations where neither of the following methods is applicable.

The S-indexing method distributes RUIs by bound values of variables in a rule, and is useful for non-conjunctive connectives[41] applied to nodes that dominate the same set of variables. The idea is to create an index key table that associates to a certain key a given subset of RUIs structures. The key is generated from the variable instantiations, through the concatenation of all them in a given order.[42]

---

[40] The pseudo-code of the MULTI procedure **schedule** is presented latter on Figure 35.

[41] According to (Choi, et al., 1992), in a non-conjunctive connective the number of antecedents needed to derive a consequent in an inference rule is not always equal to the total number of antecedents. On this definition we can fit numerical-entailment, and-or, thresh and or-entailment. And-Entailment always needs to prove all the antecedents and consequently is a conjunctive connective.

[42] That is why it is necessary that all nodes dominate the same set of variables.

For instance if we have the RUI-set

$$\left\{\begin{array}{c}(\{{}^{V1}/_{A}, {}^{V2}/_{C}\}, 2,1, \{P1: true, P2: true, P3: false\}), (\{{}^{V1}/_{A}, {}^{V2}/_{C}\}, 1,0, \{P1: true\}),\\ (\{{}^{V1}/_{B}, {}^{V2}/_{D}\}, 3,0, \{P1: true, P2: true, P3: true\})\end{array}\right\},$$

this would give rise to the S-index of Figure 29.

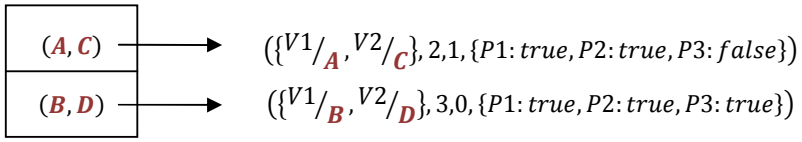


Figure 29 – S-index diagram.

It is necessary that all nodes $P1$, $P2$, $P3$ dominate uniquely variables $V1$ and $V2$ to allow correct indexing. Besides we note that the RUI that appears in the second place of the RUI-set doesn't have to be represented on the S-index since it corresponds to redundant information.

As Choi stresses, the use of S-indexing avoids the check of variable bindings. In fact after creating the index key based on the substitution of a new RUI, we always reach a consistent RUI to update.

The iteration over the RUIs on both linear and S-index methods is trivial: on the first we need to simple iterate a set and on the second we iterate the index table and for each entry we check the RUI.

P-tree method to store RUI on conjunctive rule nodes

The P-tree method arranges RUI structures on a binary tree where: "a leaf node corresponds to an antecedent, a parent node is a conjunction of its children and the root node represents the whole conjunctions of the rule premise".[43] The method is used on and-entailment and each node of the P-tree contains a set of RUIs with solutions to the conjunction of its children.

The way antecedents are placed on the binary tree depends on the number of shared variables: antecedent that share many variables are closer in the P-tree. As we will see, this property will be useful.

Figure 30 shows an example of a P-tree for rule $\{A(x,z), B(y,w), C(x,z,y), D(k)\}$ & $\Rightarrow \{E(x)\}$.

[43] (Choi, et al., 1992).

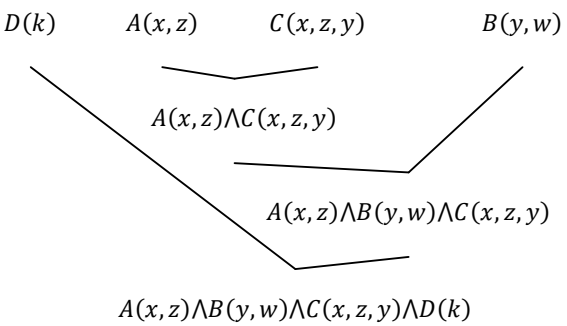


Figure 30 – P-tree for an and-entailment rule. Although not represented, attached to each node we have a set o RUIs with solutions for the conjunction of children.

Antecedents $A(x,z)$ and $C(x,z,y)$ are close to each other and have the same parent because they share two variables: x and z. $B(y,w)$ will share the same parent with the conjunction of the previous two antecedents because variable y is common. $D(k)$ doesn't share any variables with the other antecedents and consequently is the last to be conjugated.

The way RUIs are distributed in a tree-like structure suggests that, in order to iterate the set of RUIs, we should traverse the tree. We will do it in a depth-first manner. This choice has several advantages:

- if some node has any RUI associated, then all its children are necessarily solved and have a consistent solution. As we are interested in calculating the highest value $pos_{\&\Rightarrow}$ gets for all RUIs, there is no need to check deeper RUIs in the P-tree if there is some RUI for the conjunction of the children. Using the previous example, if the node $A(x,z) \wedge C(x,z,y)$ has some RUI associated, then there is certainly a consistent instantiation for both antecedents $A(x,z)$ and $C(x,z,y)$.
- the depth-first order of exploration of the top leafs of the P-tree can be interestingly used as an heuristic for the exploration of the antecedents. A depth-first exploration of the P-tree of Figure 30 would reach top leafs by the left-to-right order on the figure: $D(k) \gg A(x,z) \gg C(x,z,y) \gg B(y,w)$. As referred, leafs are arranged such that antecedents with common variables are close to each other. If antecedents are scheduled to work by this order, then the probability of failing to find a consistent instantiation and leaving the next antecedent in the idle state is considerably increased. As a comparison, let's admit that antecedents are taken from the idle mode by the following order: $D(k) \gg B(y,w) \gg A(x,z) \gg C(x,z,y)$. Using this order, variable binding conflicts may only be detected when $C(x,z,y)$ eventually reports solutions. Until then, each antecedent doesn't share any variables with the previously run antecedents.

Checking every Consequent-channel

Finally, now that we know how to iterate sets of RUIs, we note that a node may have more than one consequent channel installed.[44] The structure of a consequent-channel includes three elements:

- a node-set with the antecedents of the rule node;
- a channel structure including filter, switch and the node destination of the channel;
- a set of RUI structures represented using one of the methods previously discussed.

We must iterate all the consequent-channels and check every set of RUI structures to assure that all idle nodes that should be working are in fact set to the working state. A set of working antecedents is calculated for each consequent-channel and they are all joined to make a final set. In the end the register ***WORKING-ANTECEDENTS*** should be updated to that set.

The order of the antecedents matters!

On section 4.2.2, we warned to the fact that as opposed to what happens with PROLOG, the notion of ordering of the antecedents isn't defined on SNePS. When the processes coupled to the antecedent nodes are scheduled on the MULTI queue, there is no ordering relation between them. This situation is partially explained by the FIFO policy used on the queue and by the simultaneous insertion of all the antecedent processes. Considering that on this chapter we worked on the idleness of some processes, this problem acquires additional relevance.

Except when the P-tree method is used to store the RUI structures and the scheduling order of the antecedent is determined by the order obtained through the depth-first exploration of the tree, the other methods (linear and S-indexing) do not suggest any order on the scheduling of the antecedents. Nevertheless a proper permutation of the antecedent can minimize the number of working antecedents and leave more processes on the idle queue. Clearly if we first run the processes that will fail to find consistent solutions with higher probability, then more processes will remain idle.

Taking into account that when linear and S-indexing methods are used our implementation simply picks the first n antecedents of the list of antecedents, where n is given by the last column of Table 3, sorting that list can be beneficial to the performance of the system. However we didn't work on the implementation of multiple sorting policies.

5.5. Examples

On this section we provide some examples that show how the ideas we introduced on this chapter can reduce the search space.

[44] The register *RULE-USE-CHANNELS* stores all those channels.

5.5.1. And-entailment example: "Married persons live together!"

This example tries to conclude who lives with *Sofia*, based on the premise that all married man and woman live together and some facts about people. A detailed documentation of the entire deductions for this example can be found on Appendix 9.2. The SNePSLOG[45] file for this example can be found on Table 4 and the query made was

ask liveTogether(?who, Sofia).

```
all (x,y) ({man(x), woman(y), married(x,y)} \&=> {liveTogether(x,y)}).
Woman(Maria).
Woman(Teresa).
Woman(Joana).
Woman(Ines).
Woman(Sofia).
Man(Pedro).
Man(Gustavo).
Man(Emanuel).
Man(Jose).
Man(Afonso).
Man(Joao).
Married(Pedro, Maria).
Married(Jose, Teresa).
Married(Emanuel, Joana).
Married(Afonso, Ines).
```

Table 4 - SNePSLOG input file for the "Married Man and Woman Live Together!" example.

The example is intentionally simple and has one single rule:

$$\{Man(x), Woman(y), Married(x,y)\} \,\&\Rightarrow \{LiveTogether(x,y)\}.$$

Given that an and-entailment rule is used, the P-tree method is adopted to store reports from the antecedents. The P-tree created for this example is shown on Figure 31.

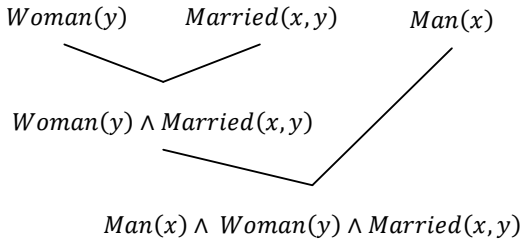


Figure 31 – P-tree for the antecedents of the rule.

[45] SNePSLOG is a software package that comes with SNePS, which translates logical formulas into a semantic network representation. From now on instead of using SNePSUL language, we will exclusively use SNePSLOG. This language is closer to first order logical formulas and can be more easily understandable. Predicates of the form $Rel(x_1, \ldots, x_n)$ are converted into pattern nodes with an arc *r* pointing to the constant node Rel and arcs *a1,…, an* pointing to corresponding arguments of the predicate (this representation is very close to the one we have been using so far). Higher order logical formulas are obviously accepted.

The order for the exploration of the antecedents is, as we discussed, given by depth-first search of the tree: $Woman(y) \gg Married(x,y) \gg Man(x)$.

When the query above is performed, the substitution $\{^{y}/_{Sofia}\}$ is applied. Then, the following sequence of events happens:

- the solution $Woman(Sofia)$ is found for the first antecedent running. The antecedent $Married(x,Sofia)$ is taken from the idle queue and set to work. $Man(x)$ remains idle;
- given that $Sofia$ is not married, no solution is found for $Married(x,Sofia)$. The deduction process ends and the antecedent $Man(x)$ is left on the idle queue not being executed. Therefore none of the facts $Man(Pedro)$, $Man(Gustavo)$, $Man(Emanuel)$, $Man(Jose)$, $Man(Afonso)$ and $Man(Joao)$ is found.

Table 5 shows the number of processes executed when the deduction is performed using and not using the IDLE queue.[46]

Type of processes executed	Using IDLE queue	Not using IDLE queue
LOW priority	8	15
Non-variable LOW priority	5	12
HIGH priority	3	15

Table 5 – Comparison of the number of processes executed using and not using the IDLE queue for the "Married man and woman live together!" example.

From the above data, it is clear that the use of the IDLE queue reduces the work performed. The considerable difference on the number of processes results from the fact that the antecedent $Man(x)$ is not executed, and therefore none of the Man are found.[47]

5.5.2. Example using And-Or node: "Does John work as a Journalist?"

This second example tries to conclude whether *John* works or not as a Journalist, based on a series of rules and facts about different jobs that can be found on Table 6. We used the SNePSLOG command

ask Job(John, Journalist)

to initiate the inference process. The entire deductions are documented on Appendix 9.3.

[46] Although we could have used metrics like the amount of time and space spent, we decided to use only the number of processes executed, as we consider that this metric makes the impact of our changes clear.

[47] The number of high priority processes can be easily calculated: $1\ user\ process + 2 \times (7\ processes\ containing\ solution) = 15$. The 7 solutions come from 6 men and 1 woman. That number of process is doubled because the propagation of the solutions to the rule node passes two channels (from the asserted solution to the antecedent pattern node and from that node to the and-entailment rule node).

```
all(x) (andor(2,3) {Job(x, Nurse), Job(x, Policeman), Job(x, Journalist), Job(x,Teacher),
                    Job(x, Investor), Job(x, Photographer), Job(x, Thief)}).
all(x) ({Has(x, PhotographicMachine), Is(x, Artistic), WorkingPlace (x,Studio)} \&=> {Job(x, Photographer)}).
all(x) ({Has(x, Seringue), Studied(x, Nursing), WorkingPlace (x,Hospital)} \&=> {Job(x, Nurse)}).
all(x) ({Has(x, LotsOfMoney)} => {Job(x, Investor)}).
all(x) ({Has(x, Dog), Lived(x, GreatAdventures)} => {Job(x, Journalist)}).
Job(DickTracy, Policeman).
Job(Heisenberg, Teacher).
Job(Einstein, Teacher).
Has(BillGates, LotsOfMoney).
Has(Tintin, Dog).
Lived(Tintin, GreatAdventures).
Has(BillGates, PhotographicMachine).
Is(Tintin, Talented).
WorkingPlace(HarrisonFord,Studio).
Has(John, PhotographicMachine).
Is(John, Artistic).
WorkingPlace(John, Studio).
```

Table 6 – SNePSLOG input file for the "Does John work as a journalist?" example.

Table 7 presents a comparison of the number of processes executed using and not using the idle queue for this example. This data should be analyzed together with Table 15 and Table 16, presented as appendixes.

Type of processes executed	Using IDLE queue	Not using IDLE queue
LOW priority	42	43
Non-variable LOW priority	22	28
HIGH priority	10	10

Table 7 - Comparison of the number of processes executed using and not using the IDLE queue for the "Does John work as a Journalist?" example.

We will focus on the rule that states that anyone has between two and three jobs:

$$\forall x \, \wedge \mathrm{V}_2^3 \left\{ \begin{matrix} Job(x, Nurse), Job(x, Policeman), Job(x, Journalist), Job(x, Teacher), \\ Job(x, Investor), Job(x, Photographer), Job(x, Thief) \end{matrix} \right\}.$$

From the previous data and from the results presented as appendixes we can conclude the following three points:

- the number of inferences made (the amount of reports propagated) is the same, as the number of HIGH priority processes executed is equal in both cases;
- the difference in the number of non-variable LOW priority processes executed is explained by the idleness of the process associated to the node P1, which is connected to the and-or rule node by an *arg* arc and represents the predicate $Job(x, Nurse)$. Table 16 presents in red color the inference steps that are avoided using the idle queue, all of them related to the *Nursing* job;
- the total number of processes executed differs only by 1 (42 + 10 using the IDLE queue and 43 + 10 not using that queue). However, as the table shows, this happens because of the execution of processes associated to variable nodes, in this case variables V1, V2, V3, V4 and V5. The idleness of some

processes had as a consequence the scheduling of all 5 variable processes one time more. Nevertheless we stress that this detail does not represent a problem because these processes perform no work and not noticing this subtlety would incur in a misunderstanding of the results.

As we noted before, the node which execution is avoided is one of the antecedents of the and-or rule written above. We now show why it is safe to idle one of the antecedents of this and-or node, applying the appropriate formula from Table 3. Given that from the SNePSLOG facts presented on Table 6, we can derive $Job(John, Photographer)$, then $pos_{\wedge\vee} = 1$. Additionally, knowing that the rule has 7 *arg* arcs, $num\_ants = 6$. Applying the appropriate formula we have

$$working\_ants_{\wedge\vee} = max\{min\{6, 6-3+1+1\}, min\{6, 2+0\}\} = 5.$$

The pattern node that represents $Job(x, Nurse)$ was the one chosen to remain idle and as a consequence the and-entailment rule

$$\forall x\{Has(x, Seringue), Studied(x, Nursing), WorkingPlace(x, Hospital)\}\& \Rightarrow \{Job(x, Nurse)\}'$$

was not triggered.

Although the quantitative results of Table 5 and Table 7 may seem modest, it is important to realize that idling some antecedents may result in pruning entire branches of the search space. On our example, if $Job(x, Nurse)$ would be the consequent of several complex rules, avoiding the execution of this process would produce more expressive results.

6. Making the Search Strategy Versatile

In section 4.2.2, “Comparing search strategies”, we discussed the differences between the search strategies used by PROLOG and by SNIP. On the first case, we explained how a depth-first backtracking search was induced on the space of SLD-refutations, by choosing an appropriate selection function and assuming the standard ordering rule. On the second case, we explained how the FIFO scheduling policy of the MULTI low priority queue resulted in a breath-first search for solutions on the network.

Also in chapter 4, we had the opportunity to compare the way each system handles recursive rules. We pointed out how it was crucial to understand the search strategy used by PROLOG to efficiently code recursive rules. Finally we discussed how the SNePS approach to recursive rules avoids the circularity of those rules.

From those arguments it is easy to understand that if we execute the same logical program in two PROLOG interpreters that implement different search strategies, then different results can emerge: while the program can stop with a solution on one case, on the other it can loop forever. PROLOG programmers must keep in their minds the search strategy that will be used to run their programs and sort program clauses and rule antecedents accordingly.

On SNIP this doesn’t seem to be a problem and having a versatile search strategy may be a luxury we can afford. If we change the scheduling policy of our MULTI queue, is the correctness of the deduction process still guaranteed? If yes, do we have any advantages on doing that?

In this chapter we explore these and other related questions. Firstly we argue that the breath first search may not be the best possible search strategy for every problem. Afterwards we discuss how to parameterize the search strategy on SNIP. Then, we discuss that searching using a heuristic may be interesting to make solutions emerge quicker and we give some hints that can be followed to create appropriate heuristics for each problem.

6.1. When breath-first may not be the best solution

Using a FIFO scheduling policy and consequently a breath-first search strategy may be appropriate if we are interested in knowing all the inferable solutions. In that case, to exhaustively traverse all the search space seems to be inescapable.

Nevertheless if our intention is to find one single solution using the smallest amount of time and space resources, then other scheduling policies can certainly do better.

We can illustrate this idea using a simple semantic network like the one of Figure 32.

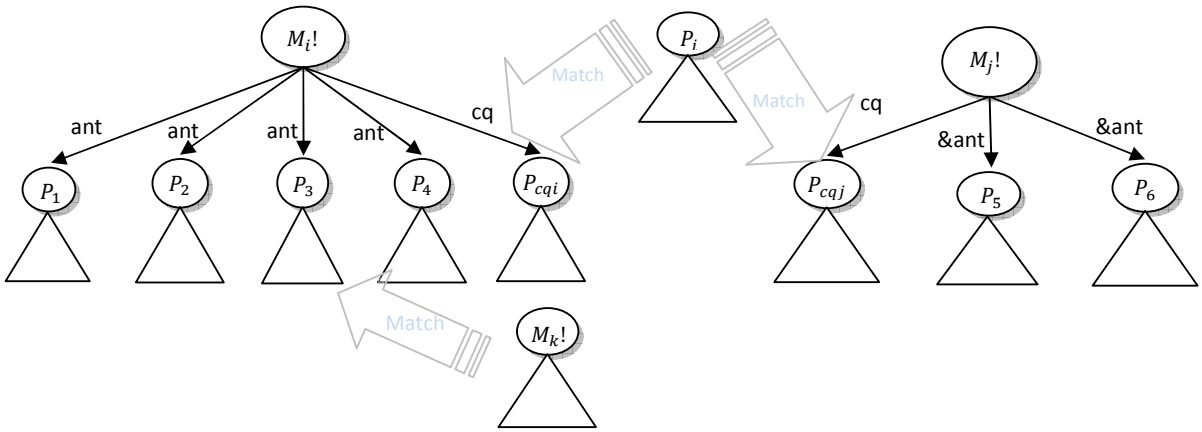


Figure 32 – Semantic network with one single solution.

This network has two rule nodes: one or-entailment with four antecedents and an and-entailment with only two. Only the antecedent P_3has the solution $M_k!$. We are interested in finding instances of P_i, taking into consideration that it matches the consequents of both rules.

Having this example in mind, it is not difficult to understand that if we use a depth-first strategy to explore the network instead of the common breath-first, then we could reach the only available solution quicker. Assuming that the or-entailment node is chosen first then, even before sending a request to the and-entailment node $M_j!$, we could search the antecedents of the or-entailment $M_i!$ and find a solution for P_{cqi} (which would be propagated to P_i). Additionally, searching the antecedent P_3before P_1, P_2 and P_4 would find a solution without running these last antecedents.

After this, we can conclude that the parameterization of the search strategy may give more control over the inference process. Besides adequate strategies can make this process more efficient and make solutions emerge earlier.

6.2. Suggesting other search strategies

The independence of the MULTI processes makes the scheduling order of new processes on the queues irrelevant to the correct operation of the inference process. On section 4.2.4, “Binding of common variables”, we gave arguments that supported this independence of behavior: we discussed how antecedents of the same rule do not communicate solutions with each other and how there is potential to parallelization of processes.

We will let the scheduling policy of the high priority queue unchanged, as we consider that it is appropriate to propagate newly found solutions. On the other hand we suggest different scheduling policies to the low priority queue, because changing the order of this queue would enable us to control the way the search space is traversed. This will have impact on the total number of times a process is scheduled and on the amount of work performed before finding a given solution.

Besides the breath-first search already implemented, we suggest two additional search methods induced by different scheduling policies for the low priority queue:

- depth-first search: using a LIFO scheduling policy for the low priority queue;
- heuristic search: assigning priorities to processes and ordering the queue according to those priorities. This can be considered as a greedy search.

6.2.1. Choosing the heuristic: what can we use to guide us?

On this section we discuss the problem of finding network criteria that can be used to create interesting heuristics to order the low priority queue.

It is not our objective to provide an heuristic that can be universally applied, always improving the performance of the system, but rather to point out some features that may be taken into account when building one. This approach can be justified by two reasons: first, there is no such thing as an heuristic that is always more efficient for all problems and second because the heuristic will necessarily depend on the structure of the network.

Comparing the potential of rule nodes to produce solutions

The amount of work that is necessary to apply an inference rule and to produce an instance of a consequent of that rule depends on two things: the cost of solving each of the antecedents and the number of antecedents that we need to solve. As an example, if we have to choose between the selection of an and-entailment and an or-entailment rule then, although the or-entailment rule only needs the solution of one of the antecedents, that antecedent may be too hard to solve and require more effort than the solution for all the antecedent of the and-entailment rule node. On this section we give a very simple metric that may be interesting to compare rule-nodes among them and sort the queue in an appropriate way.

Let's admit that each rule node has some way to estimate the cost of solving each of its antecedents and that the costs are given by an n-tuple $\langle c_1, c_2, \dots, c_n \rangle$, where c_i is the i^{th} lowest cost on the tuple, that is we assume that the costs are sorted. Then we can estimate the cost of applying the rule as Table 8 shows.

Rule	Estimated cost of solution
And-Entailment $\{A_1, \dots, A_n\}\& \Rightarrow \{C_1, \dots, C_m\}$	$\sum_{k=1}^{n} c_k$
Or-Entailment $\{A_1, \dots, A_n\}\vee \Rightarrow \{C_1, \dots, C_m\}$	c_1
Numerical Entailment $\{A_1, \dots, A_n\}i \Rightarrow \{C_1, \dots, C_m\}$	$\sum_{k=1}^{i} c_k$
And-Or $\wedge\vee_i^j\{P_1, \dots, P_n\}$	$\sum_{k=1}^{min\{j,n-i\}} c_k$

Rule	Estimated cost of solution
Thresh $\Theta_i^j\{P_1, \ldots, P_n\}$	$\sum_{k=1}^{n+i-j-1} c_k$

Table 8 - Estimated cost for obtaining a solution for a rule-node.

The estimations of the costs of the Table 8 were derived assuming that the variable instantiations produced by each antecedent are all compatible. The number of antecedent costs that are summed for each rule take into consideration the number of reports that are needed to apply each of the inference rules.

We claim that the expressions above can be related with the speed of the generation of the first solutions by each rule node and therefore may be useful when selecting the next rule node to run.

Choosing the most promising non-rule process

In this section we try to use local information provided by the network to infer how far a non-rule node is from a potential solution. Given that only local data is gathered, the estimation will obviously be rude.

Knowing that an asserted non-rule node is a solution, then it must be immediately executed as soon as it is scheduled. However when that is not the case, the number of asserted nodes in the neighborhood of the node that can potentially be matched by that node (if any) can indicate the probability of finding solutions. Additionally, the number of those potential matchers that have consequent arcs pointing to them may also hint the number of rules that may return solutions for that node. Figure 33 tries to picture these facts. Node P_i potentially matches two asserted nodes, that may become solutions, and two nodes that may be solved using rule-nodes.

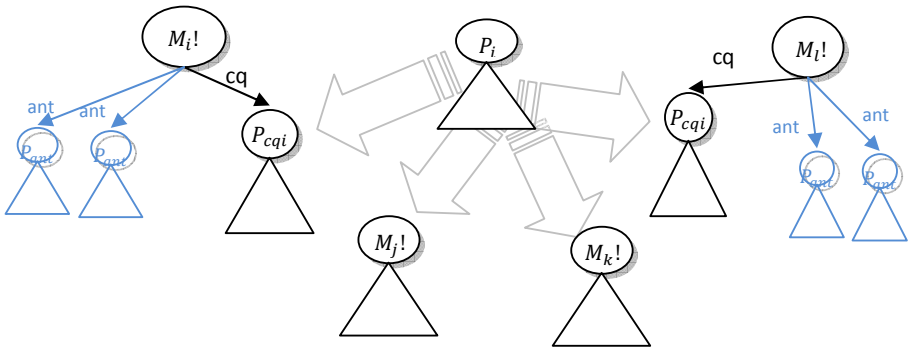


Figure 33 – Neighborhood of the non-rule node may indicate how easily it will be solved.

Instead of applying the match algorithm, which is a heavy operation, we suggest relaxing that algorithm. For that we assume the usage of the SNePSLOG predefined case-frame to the representation of predicates of the form $Rel(arg_1, \ldots, arg_n)$, which is represented on Figure 34.

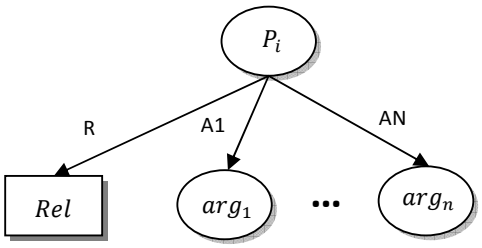


Figure 34 – Predefined case-frame to the representation of $Rel(arg_1, \ldots, arg_n)$.

This predefined case-frame and the connectivity of the network allow us to easily achieve all the nodes that share predicate Rel. For that all we need is to follow, from the node P_i, the sequence arc R/arc R- to reach all those nodes.[48] If we ignore the instantiations of the arguments of the predicate, then this operation returns a set of potential matchers for our node. The uniform traverse of the semantic network is a desirable consequence of the usage of this predefined case-frame.

Using semantic vicinity

(Schubert, et al., 1979) cite Shapiro to underline one of the most important and distinctive properties of semantic networks, that derives from the Uniqueness Principle (see section 2.2): the fact that each concept is represented only once on the network makes all the knowledge known about that concept accessible from a common place. Besides this, Schubert also considers that "the fundamental assumption in the use of semantic nets is that the knowledge required to perform an intellectual task generally lies in the semantic vicinity of the concepts involved in the task". It would be interesting to include this notion of semantic vicinity on our heuristics.

Let's admit that our network contains information about several domains of knowledge, say *politics*, *music* and *sports*. If we ask information about the *Tour De France* then we are certainly talking about sports. Near the node representing the concept *Tour De France*, we could certainly find information about *cycling* and, going a bit deeper in the network, information about other sporting activities. Cycling competitions may also be connected to *politics*, but that would imply traversing many arcs on the network. Making a breath-first search on the network graph, starting from the concepts we are willing to find information about, allow us to find other concepts that may be close in meaning.

During the deduction process and continuing under the assumption that we are interested in the *Tour De France*, let's consider that we are supposed to choose between two processes, one that is connected to a node related with the *Rolling Stones* and another to a node connected to *Football*. Which seems to be closer to a solution to our query? *Football* is a sport and the Tour De France is a sporting competition… We would certainly schedule the second process first on the low priority queue.

Incorporating this kind of information on the deduction process may be accomplished in the following way:

[48] Remember that all the nodes contain a list of the converse arcs that point that node. Therefore this is an easy task.

- when the user asks for a deduction, we start by performing a breath-first search around the concepts that are present on the query made, until reaching a constant depth. Going too depth in the search will tend to find more general and unrelated concepts to our initial ones. Then, we build a set with the concepts found on this search;
- during the inference process, if we come across concepts that belong to the set built in the beginning, then this may hint that we are closer to a solution. We can use this information to schedule the process in the beginning of the MULTI queue.

What can we expect from these heuristics?

As we already referred, the applicability of these ideas is not universal. The ideas explained on this section use local network information to infer the proximity of solutions. On complex networks, containing solutions at different depths and hundreds of nodes, this may not be enough.

Global properties can be obtained preprocessing the network. This may provide more accurate information about solutions, but may imply a great deal of computational effort. Additionally, the SNePS semantic networks are mutable, as new nodes are either asserted by the user or inferred by SNIP. This also causes problems on the calculation and update of metrics that can be used by heuristics.

The work of (Smith, 1989) is a reference on the control of backward inference and may provide interesting ideas on this domain. This topic can be explored as future work.

6.3. Implementation of the solution

To implement the possibility of having different strategies we made the following changes to the SNePS' code:

- we created a new command on SNePS with the following syntax:

  (setstrategy *type-of-strategy [heuristic-number]*)

  where ***type-of-strategy*** is **FIFO**, **LIFO** or **PRIORITY** and ***heuristic-number*** indicates, for the last strategy, the number of the heuristic to use (see latter). This command sets the created global variables ***search-strategy*** and ***heuristic*** to the appropriate values;
- we added two registers to the MULTI processes, ***PRIORITY-CALCULATOR*** and ***QUEUE-PRIORITY***: the first register contains a function that receives a process and returns an integer number that corresponds to the priority of that process in the queue; the second stores that priority number and is used to keep the queue sorted;
- we changed the MULTI procedure **schedule**, which is responsible for the insertion of new processes on the queues. The simplified pseudo-code of this procedure is shown in Figure 35.[49]

[49] The pseudo-code presented here was written with the intention of making our changes clear. Some details related with other packages of SNePS not mentioned on this work were avoided.

```
function schedule (process, queue)
// Returns the queue with the process inserted
global vars:    *search-strategy*, // FIFO, LIFO or PRIORITY strategies. This variable is set by the setstrategy SNePS command
{
        if ("process is already on the queue") {
                return queue;
        }
        elseIf("process is an USER process") {
                insert-front(process, queue);
        }
        elseIf("process has low priority")
                insert-policy(process, queue, *search-strategy*);
        }
        else { // high priority processes are scheduled as they were in the past
                insert-rear(process, queue);
        }
        return queue;
}
```

Figure 35 – Pseudo-code of the new MULTI procedure schedule.

The function **insert-policy** inserts the process in the queue according to the search-strategy selected: either in the beginning of the queue, in the end or setting ***QUEUE-PRIORITY*** to the integer returned by the ***PRIORITY-CALCULATOR*** function and keeping the queue sorted by that number;

- we created the structure **heuristic** that stores on each field different ***PRIORITY-CALCULATOR***s, one for each node type. The global variable ***heuristic*** keeps a structure of that type. When a new process is created, the register ***PRIORITY-CALCULATOR*** is set to the function that is indicated on the variable ***heuristic***, according to the node type;
- finally we created a global variable, ***available-heuristics***, containing a list of possible **heuristic**s. The number indicated in the command **setstrategy** corresponds to the order number on this list.

6.4. An example

To show the impact of using multiple search strategies on the number of processes executed, we created the example described on the appendix 9.4. The idea is to conclude which magazine or newspaper would publish the *Tour De France* event, considering that press is specialized on one topic such as sports, politics or music. The SNePSLOG commands can be found on subsection 9.4.1.

The heuristic used to obtain the results of the tables below is very simple and has the following properties:

- the priority of the rule node processes was estimated applying the functions of Table 8 and assuming equal cost for all the antecedents (we used a constant integer cost of 2 for each antecedent);
- non-rule node processes were assigned priorities such that the following relative ordering among these processes is obtained:
  - first we find the processes that have semantic proximity with the initial query. This semantic proximity was measured like we suggested on subsection "Using semantic";
  - then we find the processes that are associated to asserted nodes, which may become solutions;

- next we have the non-asserted node processes that can potentially match asserted nodes (see subsection ”Choosing the most promising non-rule process”);
- after these, we find the nodes that can potentially match non-asserted nodes that may be proved by some rule, that is that have consequent channels pointing them;
- finally we have all the remaining non-rule processes.

The results obtained on Table 9 show the total number of processes executed for each search strategy. Clearly the FIFO strategy seems to be the strategy that obtains the lowest number of executed processes.

Type of processes executed	FIFO	LIFO	HEURISTIC
LOW priority	162	287	166
Non-variable LOW priority	66	70	70
HIGH priority	93	102	102

Table 9 – Total number of processes executed for the example “Which press publishes the Tour De France?”.

Nevertheless, if instead of the total number of processes run during the entire deduction process we check the number of processes executed until reaching the solution, we conclude that the heuristic search can outperform both FIFO and LIFO strategies.

Type of processes executed	FIFO	LIFO	HEURISTIC
LOW priority	157	281	75
Non-variable LOW priority	62	69	51
HIGH priority	81	100	76

Table 10 – Number of processes executed until reaching the first solution, using the same example.

As we discussed on subsection “What can we expect from these heuristics?”, the applicability of the heuristic used here on more complex networks is doubtful.

The good results obtained on this particular example can be partially explained by the small size of the network and by the fact that solutions appear more or less at the same depth on the search tree.

7. Conclusion

On this work, we started by describing the SNePS semantic network and the node-based backward inference implemented by SNIP. Afterwards, we presented SLD-resolution as a refinement of the original Resolution mechanism and explained how PROLOG searches for SLD-refutations using a depth-first backtracking search. Then we presented a brief comparison between both automatic deduction systems, with the intention of familiarizing the reader with the advantages of each mechanism and finding ideas that could be applied to improve and control the inference performed by SNIP.

After this theoretical effort, we used some of the ideas that emerged from our comparison to suggest improvements to SNePS. We started by observing that SNIP worked on all the antecedents of rules, even when that was unnecessary, that is when there was no hope to apply the inference rule. This was shocking for And-entailment rule node: even when one antecedent didn't produce any solutions, all the antecedents were executed. This is not acceptable when efficiency is a priority.

To solve this problem, we begun by deriving conditions that express the number of antecedents that should work for each rule node and which depend of the number of true and false solutions obtained for the antecedents of that node. We discussed possible implementations, we elected one and we implemented it. Finally we provided examples showing that the number of processes executed was reduced and that some branches of the search tree were avoided without sacrificing the total number of solutions.

In a second phase of our work, motivated by the fact that MULTI processes run independently and that the search strategies were different on each of the systems we analyzed, we decided to allow the user to freely parameterize the search strategy used on SNIP. We argued that this may be desirable and confers control over the inference mechanism to the user.

To accomplish the task we changed the scheduling policy of the low priority MULTI queue and allowed the creation of heuristics to calculate the priorities of the processes.

Further work can be suggested on two fields. First, we suggest the implementation of a multi-threading MULTI system.[50] The current version of MULTI is somehow deceiving in the sense that it does not implement multiprocessing at all. Finally, the development of more accurate heuristics based on the global analysis of the network is a natural extension of this work.

[50] I thank Professor Ernesto Morgado for suggesting this idea that was unfortunately not explored.

8. References

Brachman, J. Ronald. 1979. On the Epistemological Status of Semantic Networks. [ed.] Nicholas V. Findler. *Associative Networks, Representation and Use of Knowledge by Computers.* s.l. : Academic Press, 1979, 1, pp. 3-50.

Brachman, R. J. and Schmolze, J. G. 1985. An Overview of the KL-ONE knowledge representation system. *Cognitive Science.* April 1985, Vol. 9, pp. 171-216.

Choi, Joongmin and Shapiro, Stuart C. 1992. Efficient Implementation of Non-Standard Connectives and Quantifiers in Deduction Reasoning Systems. 1992.

Collins, A. M. and Quillian, M. R. 1969. Retrieval Time from Semantic Memory. *Journal of Verbal Learning and Verbal Behaviour.* 1969, Vol. 8, pp. 240-248.

Cravo, Maria dos Remédios and Martins, João Pavão. 1989. Path-Based Inference in SNeBR. [ed.] João Pavão Martins and Ernesto Morgado. *EPIA 89 - 4th Portuguese Conference on Artificial Inteligence.* Lisbon : Springer-Verlag, 1989, Vol. 390, pp. 97-106.

Cruz-Filipe, Luís. 2006. *Programação em Lógica.* IST : Unpublished lecture notes, 2006.

Davis, M. and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *Automation of Reasoning: Classical Papers on Computational Logic 1957-1966.* 1st Edition. s.l. : Springer, 1960, Vol. 1, pp. 125-139.

Doyle, J. 1979. A Truth Maintenance System. *Artificial Intelligence.* 1979, Vol. 12, pp. 231-272.

Hull, Richard G. 1986. *A New Design for SNIP, the SNePS Inference Package.* Department of Computer Science, State University of New York at Buffalo. 1986. 14.

Kowalski, R. and Kuehner, D. 1983. Linear Resolution with Selection Function. [book auth.] Jorg Siekmann and Graham Wrightson. *Automation of Reasoning: Classical Papers on Computational Logic.* s.l. : Springer, 1983, Vol. 2.

Lehmann, Fritz. 1992. *Semantic Networks in Artificial Intelligence.* s.l. : Pergamon Press Ltd, 1992.

Levesque, Hector and Mylopoulos, John. 1979. A Procedural Semantics for Semantic Networks. [book auth.] Nicholas V. Findler. *Associative Networks: Representation and use of Knowledge by Computers.* s.l. : Academic Press, Inc., 1979, pp. 93-120.

Loveland, Donald W. 1978. *Automated Theorem Proving: a logical basis.* 1st Edition. s.l. : North-Holland Publishing Company, 1978.

Mamede, Nuno J. N. 1992. *A Lógica de LORE e o Raciocínio Abdutivo Não Omnisciente.* Lisboa : s.n., 1992. pp. 145 - 188.

Martins, João P. and Shapiro, Stuart C. 1983. Reasoning in Multiple Belief Spaces. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* 1983, pp. 370 - 373.

Martins, João Pavão. 2005. Semantic Networks. *Knowledge Representation.* Unpublished Lecture Notes. 2005, 6, pp. 213-267.

McCune, Willian and Wos, L. 1997. The CADE-13 competition incarnations. *Journal of Automated Reasoning.* 1997, Vol. 18(2), pp. 211-220.

McKay, Donald P. and Shapiro, Stuart C. 1980. *MULTI - A LISP Based Multiprocessing System.* Department of Computer Science, State University of New York at Buffalo. 1980.

—. 1981. Using Active Connection Graphs for Reasoning with Recursive Rules. 1981, pp. 368-374.

Pearl, Judea. 1988. Probabilistic Reasoning in Intelligent Systems. 1988.

Peirce, Charles Sanders. 1980. On the algebra of logic. *American Journal of Mathematics.* 1980, Vol. 3, pp. 15-57.

Robinson, J. A. 1965. A Machine Oriented Logic Based on the Resolution Principle. [ed.] Jorg Siekmann and Graham Wrightson. *Automation of Reasoning: Classical Papers on Computational Logic 1957 - 1966.* 1965, Vol. 1, pp. 397-415.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. 1986. Learning internal representation by error propagation. [book auth.] D. E. Rumelhart, J. L. McClelland and The PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge : MIT Press, 1986, Vol. 1, pp. 318-362.

Schank, Roger C. and Tesler, Larry G. 1969. A conceptual parser for natural language. *Proc. IJCAI-69.* 1969, pp. 569-578.

Schubert, Lenhart K., Goebel, Randolph G. and Cercone, Nicholas J. 1979. The Structure and Organization of a Semantic Net for Comprehension and Inference. [ed.] Nicholas V. Findler. *Associative Networks: Representation and Use of Knowledge by Computers.* s.l. : Academic Press, Inc., 1979, pp. 121-175.

Shapiro, Stuart C. and Group, The SNePS Implementation. 2004. *SNePS 2.6.1 User's Manual.* 2004.

Shapiro, Stuart C. and McKay, Donald P. 1980. Inference with Recursive Rules. *Proc. NCAI.* 1980, pp. 151-153.

Shapiro, Stuart C. and Neal, J. G. 1982. A knowledge engineering approach to natural language understanding. *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics.* 1982, pp. 136-144.

Shapiro, Stuart C. and Rapaport, William J. 1987. SNePS Considered as a Fully Intensional Propositional Semantic Network. [ed.] Nick Cercone and Gordon McCalla. *The Knowledge Frontier: Essays in the Representation of Knowledge.* s.l. : Springer-Verlag, 1987, pp. 262-315.

—. 1992. The SNePS Family. *Computers & Mathematics with Applications.* January-March 1992, Vols. 23(2-5), pp. 243-275.

Shapiro, Stuart C. 1978. Path-Based and Node-Based Inference in Semantic Networks. 1978.

—. 1977. Representing and Locating Deduction Rules in a Semantic Network. *SIGART Newsletter.* 1977, Vol. 63, pp. 14-18.

—. 1999. SNePS: A Logic for Natural Language Understanding and Commonsense Reasoning. *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language.* 1999.

—. 1979. The SNePS semantic network procesing system. *Associative Networks.* 1979, pp. 179-203.

—. 1979. Using Non-Standard Connectives and Quantifiers for Representing Deduction Rules in a Semantic Network. 1979.

Shapiro, Stuart C., et al. 1981. *SNePSLOG - A "Higher Order" Logic Programming Language.* Department of Computer Science, State University of New York at Buffalo. Amherst, NY : s.n., 1981.

Smith, David E. 1989. Controlling Backward Inference. *Artificial Intelligence.* 1989, Vol. 39, pp. 145-208.

Socher-Ambrosius, Rolf and Johann, Patricia. 1997. *Deduction Systems.* 1st Edition. s.l. : Springer, 1997.

Sowa, John F. 1992. *Encyclopedia of Artificial Intelligence.* [ed.] Stuart Shapiro. 2nd edition. s.l. : Wiley, 1992.

—. 1991. *Principles of Semantic Networks.* s.l. : Morgan Kaufmann Publishers, Inc., 1991.

—. 1978. Semantics of Conceptual Graphs. 1978.

Tesnière, Lucien. 1959. Éléments de Syntaxe Structurale. 2nd edition, 1959.

9. Appendixes

9.1. Ancestors Example

This appendix contains the entire deduction process generated by SNePS, when the command

(deduce rel ancestor arg1 $who1 arg2 $who2)

is used to derive all the solutions for $Ancestor(x, y)$, based on the network of Figure 22. This example was briefly explained on section 4.2.5.

9.1.1. The SNEPSUL input file

To create the network with the $Ancestor$ relation we used the code of Table 11.

```
(ev-trace)

(define rel arg1 arg2)

(assert rel ancestor
        arg1 a
        arg2 b)

(assert rel ancestor
        arg1 b
        arg2 c)

(assert forall ($x $y $z)
        &ant ((build rel ancestor arg1 *x arg2 *y)
              (build rel ancestor arg1 *y arg2 *z))
        cq (build rel ancestor arg1 *x arg2 *z))

(deduce rel ancestor arg1 $who1 arg2 $who2)
```

Table 11 – SNePSUL code to produce the semantic networks on Figure 22.

9.1.2. A Commented Deduction Process

```
SNEPS(4): (sneps)

  Welcome to SNePS-2.6 [PL:1a 2004/08/26 23:05:27]

Copyright (C) 1984--2004 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

  6/23/2007 17:19:05

* (intext "C:\\Documents and Settings\\Pedro Neves\\My Documents\\TFC\\exemplos
sneps\\ancestors.txt")

File C:\Documents and Settings\Pedro Neves\My Documents\TFC\exemplos sneps\ancestors.txt is now
the source of input.

 CPU time : 0.00

*
(REL ARG1 ARG2)
```

CPU time : 0.00 * **(M1!)** **CPU time : 0.00** * **(M2!)** **CPU time : 0.00** * **(M3!)** **CPU time : 0.00** *	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: () >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P4 *NAME* = SNIP::NON-RULE **I wonder if** **((P4 (ARG1 V4) (ARG2 V5) (REL (ANCESTOR))))** **holds within the BS defined by context DEFAULT-DEFAULTCT** >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P4 *NAME* = SNIP::NON-RULE	Node P4 receives a request from the USER process. This node is created by the command **deduce**. Node P4 unifies with variable nodes and with molecular asserted nodes M1! M2! (which are solutions), with the antecedents of the rule P1 and P2 and finally with the consequent of the rule P3 (see step 1)
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p2| MULTI::|p3| MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p2 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V4 . A) (V5 . B)) DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M1! *NAME* = SNIP::NON-RULE **I know** **((M1! (ARG1 (A)) (ARG2 (B)) (REL (ANCESTOR))))** >>>>>> Leaving process id = p2 with bindings:	The request sent by P4 is handled and as a consequence the solution $\left\{{}^{V4}/_{a}, {}^{V5}/_{b}\right\}$ is found. The solution is send back to process P4, which is scheduled on the high priority queue (step 2).

PRIORITY = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . A) (V5 . B)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M1! *NAME* = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M1!) (V4 . A) (V5 . B)) (HYP (C2)) SNIP::POS M1! M1! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M1!) (V4 . A) (V5 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	P4 receives the solution $\{^{V4}/_{a}, ^{V5}/_{b}\}$ as a report… …stores it on the KNOWN-INSTANCES register…
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p0|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 0 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = NIL *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS P4 M1! DEFAULT-DEFAULTCT)) *NAME* = SNIP:USER >>>>>> Leaving process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 1 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL	…and sends it to the USER process. The number of solutions found is tested. However, the user did not impose any restriction on the number of desired solutions, meaning that he wants all of

DEDUCED-NODES = (M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = NIL *NAME* = SNIP:USER	them. Consequently, the inference process continues.
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p3| MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p3 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V4 . B) (V5 . C)) DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M2! *NAME* = SNIP::NON-RULE **I know** **((M2! (ARG1 (B)) (ARG2 (C)) (REL (ANCESTOR))))** >>>>>> Leaving process id = p3 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . B) (V5 . C)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M2! *NAME* = SNIP::NON-RULE	Solution $\{V4/_b, V5/_c\}$ is found and sent to the consumer / requestor P4, which is once again scheduled on the high priority queue (see step 2)
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: (MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M2!) (V4 . B) (V5 . C)) (HYP (C3)) SNIP::POS M2! M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P4 . M1!) (V4 . A) (V5 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M2!) (V4 . B) (V5 . C)) (HYP (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	Solution $\{V4/_b, V5/_c\}$ is received and propagated to…
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p0|), L: (MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|)	

>>>>>> Entering process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 1 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = ((((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS P4 M2! DEFAULT-DEFAULTCT)) *NAME* = SNIP:USER >>>>>> Leaving process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = NIL *NAME* = SNIP:USER	… the user process which… …adds one to the number of positive instances found and… … stores it in the register DEDUCED-NODES
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p4| MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p4 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((((V1 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V1 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p4 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((((V1 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V1 *NAME* = SNIP::NON-RULE	Atomic variable node V1 gets into action but no solution is found. The same happens…
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p5| MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p5 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((((V2 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V2 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p5 with bindings:	… with variable node V2…

PRIORITY = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((((V2 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V2 *NAME* = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p6| MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p6 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((((V3 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V3 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p6 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((((V3 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V3 *NAME* = SNIP::NON-RULE	… and variable node V3.
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p7| MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p7 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P1 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p7 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P1 *NAME* = SNIP::NON-RULE	(see step 3)
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p8| MULTI::|p9|) >>>>>> Entering process id = p8 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL	(see step 3)

KNOWN-INSTANCES = NIL *NODE* = P2 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p8 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P2 *NAME* = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p9|) >>>>>> Entering process id = p9 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V4 . V1) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P3 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p9 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V1) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT M3! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P3 *NAME* = SNIP::NON-RULE	Node P3 has a consequent arc from node M3! pointing to it. Therefore the request received from P4 is handled establishing a channel to that node and sending a request. (see step 4)
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p10|) >>>>>> Entering process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = NIL *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL NIL DEFAULT-DEFAULTCT P3 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE >>>>>> Leaving process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) NIL))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN)) *REQUESTS* = NIL	M3! sends requests to the antecedents of the node, P1 and P2. (see step 5)

REPORTS = NIL *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE	
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p8| MULTI::|p7|) >>>>>> Entering process id = p8 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P2 *NAME* = SNIP::NON-RULE **I wonder if** **((P2 (ARG1 V2) (ARG2 V3) (REL (ANCESTOR))))** **holds within the BS defined by context DEFAULT-DEFAULTCT** >>>>>> Leaving process id = p8 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P2 *NAME* = SNIP::NON-RULE	P2 receives the request from the rule node M3! and matches itself on the network. It establishes a channel to the other antecedent P1 and to M1! and M2! which are solutions. **Note that no channel is established to the consequent P3.** Consequently, solutions are not reiterated to the antecedents of the rule.
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p7| MULTI::|p2| MULTI::|p3| MULTI::|p4|) >>>>>> Entering process id = p7 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL NIL DEFAULT-DEFAULTCT M3! OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = P1 *NAME* = SNIP::NON-RULE **I wonder if** **((P1 (ARG1 V1) (ARG2 V2) (REL (ANCESTOR))))** **holds within the BS defined by context DEFAULT-DEFAULTCT** >>>>>> Leaving process id = p7 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL	The same happens to the antecedent P1. **Note that P1 does not match the consequent node P3 too.**

KNOWN-INSTANCES = NIL *NODE* = P1 *NAME* = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p2| MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p2 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . A) (V5 . B)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V1 . A) (V2 . B)) DEFAULT-DEFAULTCT P1 OPEN) (NIL ((V3 . B) (V2 . A)) DEFAULT-DEFAULTCT P2 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M1! *NAME* = SNIP::NON-RULE **I know** **((M1! (ARG1 (A)) (ARG2 (B)) (REL (ANCESTOR))))** >>>>>> Leaving process id = p2 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V3 . B) (V2 . A)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V1 . A) (V2 . B)) DEFAULT-DEFAULTCT P1 OPEN) (NIL ((V4 . A) (V5 . B)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M1! *NAME* = SNIP::NON-RULE	M1! sends solutions to the antecedents P1 and P2.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p7| MULTI::|p8|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p7 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P1 . M1!) (V1 . A) (V2 . B)) (HYP (C2)) SNIP::POS M1! M1! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = P1 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p7 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P1 . M1!) (V1 . A) (V2 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P1 *NAME* = SNIP::NON-RULE	P1 receives the report from M1! and forwards it not only to the rule node M3! but also to the brother P2 and to the node P4.

>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p8| MULTI::|p10| MULTI::|p1|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN)) *REPORTS* = ((((P2 . M1!) (V3 . B) (V2 . A)) (HYP (C2)) SNIP::POS M1! M1! DEFAULT-DEFAULTCT) (((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = P2 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS)) *NODE* = P2 *NAME* = SNIP::NON-RULE	Consequently P2 receives the same report **twice** and forwards it to M3! and P4.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p10| MULTI::|p1|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) NIL))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P1 . M1!) (V1 . A) (V2 . B)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT) (((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS P2 M1! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE >>>>>> Leaving process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) ((P2 (((P2 . M1!) (V3 . B) (V2 . A)) 1 0 ((P2 (DER (C2)) SNIP::TRUE)) NIL)) (P1 (((P1 . M1!) (V1 . A) (V2 . B)) 1 0 ((P1 (DER (C2)) SNIP::TRUE)) NIL)))))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-	Reports arrive at M3! M3! tries to apply and-entailment deduction rule but there is not enough solutions.

DEFAULTCT P2 OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS P2 M1! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P4 . M2!) (V4 . B) (V5 . C)) (HYP (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS) (((P4 . M2!) (V4 . B) (V5 . C)) (HYP (C3)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	P4 receives the solution $\{{}^{V4}/_{a}, {}^{V5}/_{b}\}$, **duplicated** from both P1 and P2, which is **already known**.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p0|), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS P4 M1! DEFAULT-DEFAULTCT)) *NAME* = SNIP:USER >>>>>> Leaving process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL	

TOTAL-DESIRED = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = NIL *NAME* = SNIP:USER	
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p3| MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p3 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . B) (V5 . C)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = ((NIL ((V1 . B) (V2 . C)) DEFAULT-DEFAULTCT P1 OPEN) (NIL ((V3 . C) (V2 . B)) DEFAULT-DEFAULTCT P2 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M2! *NAME* = SNIP::NON-RULE **I know** **((M2! (ARG1 (B)) (ARG2 (C)) (REL (ANCESTOR))))** >>>>>> Leaving process id = p3 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V3 . C) (V2 . B)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V1 . B) (V2 . C)) DEFAULT-DEFAULTCT P1 OPEN) (NIL ((V4 . B) (V5 . C)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M2! *NAME* = SNIP::NON-RULE	Asserted node M2! informs nodes P1 and P2 of its existence and schedules them on the high priority MULTI queue.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p7| MULTI::|p8|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p7 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P1 . M2!) (V1 . B) (V2 . C)) (HYP (C3)) SNIP::POS M2! M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P1 . M1!) (V1 . A) (V2 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P1 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p7 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN) (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P1 . M2!) (V1 . B) (V2 . C)) (HYP (C3)) SNIP::POS) (((P1 . M1!) (V1 . A) (V2 . B)) (HYP (C2)) SNIP::POS)) *NODE* = P1	P1 receives solution $\{{}^{V1}/_{b}, {}^{V2}/_{c}\}$. It is sent, similarly to what happened before, to rule-node M3! and brother P2.

NAME = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p8| MULTI::|p10| MULTI::|p1|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN)) *REPORTS* = ((((P2 . M2!) (V3 . C) (V2 . B)) (HYP (C3)) SNIP::POS M2! M2! DEFAULT-DEFAULTCT) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS)) *NODE* = P2 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS)) *NODE* = P2 *NAME* = SNIP::NON-RULE	Consequently P2 is informed of the solution $\{V2/_b, V3/_c\}$ **twice again**!
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p10| MULTI::|p1|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) ((P2 (((P2 . M1!) (V3 . B) (V2 . A)) 1 0 ((P2 (DER (C2)) SNIP::TRUE)) NIL)) (P1 (((P1 . M1!) (V1 . A) (V2 . B)) 1 0 ((P1 (DER (C2)) SNIP::TRUE)) NIL)))))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P1 . M2!) (V1 . B) (V2 . C)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS P2 M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE **Since** **((M3! (FORALL (V3 <-- C) (V2 <-- B) (V1 <-- A))** **(&ANT (P2 (ARG1 (V2 <-- B)) (ARG2 (V3 <-- C)) (REL (ANCESTOR)))** **(P1 (ARG1 (V1 <-- A)) (ARG2 (V2 <-- B)) (REL (ANCESTOR))))** **(CQ (P3 (ARG1 (V1 <-- A)) (ARG2 (V3 <-- C)) (REL (ANCESTOR))))))** **and** **((P2 (ARG1 (V2 <-- B)) (ARG2 (V3 <-- C)) (REL (ANCESTOR))))** **and**	Naturally M3! receives two reports from the antecedents: from P1 it receives $\{V1/_b, V2/_c\}$ and from P2 it receives $\{V2/_b, V3/_c\}$ At this moment, M3! knows the existence of four instances: → $\{V1/_a, V2/_b\}$ → $\{V1/_b, V2/_c\}$ → $\{V2/_a, V3/_b\}$

((P1 (ARG1 (V1 <-- A)) (ARG2 (V2 <-- B)) (REL (ANCESTOR)))) **I infer** **((P3 (ARG1 (V1 <-- A)) (ARG2 (V3 <-- C)) (REL (ANCESTOR))))** >>>>>> Leaving process id = p10 with bindings: *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *INTRODUCTION-CHANNELS* = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) (((P2 P1) (((P1 . M1!) (V1 . A) (P2 . M2!) (V3 . C) (V2 . B)) 2 0 ((P2 (DER (C3)) SNIP::TRUE) (P1 (DER (C2)) SNIP::TRUE)) NIL)) (P2 (((P2 . M1!) (V3 . B) (V2 . A)) 1 0 ((P2 (DER (C2)) SNIP::TRUE)) NIL) (((P2 . M2!) (V3 . C) (V2 . B)) 1 0 ((P2 (DER (C3)) SNIP::TRUE)) NIL)) (P1 (((P1 . M1!) (V1 . A) (V2 . B)) 1 0 ((P1 (DER (C2)) SNIP::TRUE)) NIL) (((P1 . M2!) (V1 . B) (V2 . C)) 1 0 ((P1 (DER (C3)) SNIP::TRUE)) NIL)))))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE	→ $\{^{V2}/_{b}, ^{V3}/_{c}\}$ A solution can be inferred with the binding $\{^{V1}/_{a}, ^{V2}/_{b}, ^{V3}/_{c}\}$. To understand the RULE-USE-CHANNELS data structure we suggest (Choi, et al., 1992). The new solution is sent to the consequentP3.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1| MULTI::|p9|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT) (((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS P2 M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS) (((P4 . M2!) (V4 . B) (V5 . C)) (HYP (C3)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	P4 is informed of the solution $\{^{V4}/_{b}, ^{V5}/_{c}\}$, still from the antecedents P1 and P2.
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p0| MULTI::|p9|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p0 with bindings:	

PRIORITY = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = ((((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS P4 M2! DEFAULT-DEFAULTCT)) *NAME* = SNIP:USER >>>>>> Leaving process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = NIL *NAME* = SNIP:USER	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p9|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p9 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V1) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT M3! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P3 . M4!) (V3 . C) (V1 . A)) (DER (C6)) SNIP::POS M3! M4! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = NIL *NODE* = P3 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p9 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V4 . V1) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT M3! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P3 . M4!) (V3 . C) (V1 . A)) (DER (C6)) SNIP::POS)) *NODE* = P3 *NAME* = SNIP::NON-RULE	The consequent P3 receives from the rule node M3! the new inferred solution $\{^{V1}/_{a}, ^{V3}/_{c}\}$. This solution is sent to the temporary node P4....
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M4!) (V4 . A) (V5 . C)) (DER (C6)) SNIP::POS P3 M4! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS))	... which receives it! This solution was not yet known by this node and it is immediately sent

NODE = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M4!) (V4 . A) (V5 . C)) (DER (C6)) SNIP::POS) (((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	to…
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p0|), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 2 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = ((((P4 . M4!) (V4 . A) (V5 . C)) (DER (C6)) SNIP::POS P4 M4! DEFAULT-DEFAULTCT)) *NAME* = SNIP:USER >>>>>> Leaving process id = p0 with bindings: *PRIORITY* = SNIP:HIGH *CLIENT* = SNIP:USER *NEG-FOUND* = 0 *POS-FOUND* = 3 *NEG-DESIRED* = NIL *POS-DESIRED* = NIL *TOTAL-DESIRED* = NIL *DEDUCED-NODES* = (M4! M2! M1!) *CONTEXT-NAME* = DEFAULT-DEFAULTCT *REPORTS* = NIL *NAME* = SNIP:USER	…the USER process and… … POS-FOUND is incremented by one.
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p4| MULTI::|p6| MULTI::|p8|) >>>>>> Entering process id = p4 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((((V1 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL *REQUESTS* = ((((V1 . P2)) NIL DEFAULT-DEFAULTCT P2 OPEN)) *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = V1 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p4 with bindings: *PRIORITY* = SNIP:LOW *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((((V1 . P2)) NIL DEFAULT-DEFAULTCT P2 OPEN) (((V1 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = NIL	

```
    *REQUESTS* = NIL
    *REPORTS* = NIL
    *KNOWN-INSTANCES* = NIL
    *NODE* = V1
    *NAME* = SNIP::NON-RULE
```

```
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p6| MULTI::|p8|)

>>>>>> Entering process id = p6 with bindings:

    *PRIORITY* = SNIP:LOW
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* = ((((V3 . P4)) NIL DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* = NIL
    *REQUESTS* = ((((V3 . P1)) NIL DEFAULT-DEFAULTCT P1 OPEN))
    *REPORTS* = NIL
    *KNOWN-INSTANCES* = NIL
    *NODE* = V3
    *NAME* = SNIP::NON-RULE

>>>>>> Leaving process id = p6 with bindings:

    *PRIORITY* = SNIP:LOW
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* = ((((V3 . P1)) NIL DEFAULT-DEFAULTCT P1 OPEN) (((V3 . P4)) NIL
DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* = NIL
    *REQUESTS* = NIL
    *REPORTS* = NIL
    *KNOWN-INSTANCES* = NIL
    *NODE* = V3
    *NAME* = SNIP::NON-RULE
```

```
>>>>>> MULTI QUEUES> A: (), H: (), L: (MULTI::|p8|)

>>>>>> Entering process id = p8 with bindings:

    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* =
                    ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN)
                     (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* =
               (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN)
                ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN))
    *REQUESTS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN))
    *REPORTS* = NIL
    *KNOWN-INSTANCES* =
                  ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS)
                   (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS))
    *NODE* = P2
    *NAME* = SNIP::NON-RULE

>>>>>> Leaving process id = p8 with bindings:

    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* =
                    ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN)
                     (NIL NIL DEFAULT-DEFAULTCT M3! OPEN)
                     (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* =
              (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN)
               ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN))
    *REQUESTS* = NIL
    *REPORTS* = NIL
    *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS)
                   (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS))
    *NODE* = P2
    *NAME* = SNIP::NON-RULE
```

```
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p7|), L: ()

>>>>>> Entering process id = p7 with bindings:
```

```
    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN)
                          (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN)
                          (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* =
                    (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN)
                    ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN))
    *REQUESTS* = NIL
    *REPORTS* =
                ((((P1 . M1!) (V1 . A) (V2 . B)) (DER (C2)) SNIP::POS P2 M1! DEFAULT-DEFAULTCT)
                 (((P1 . M2!) (V1 . B) (V2 . C)) (DER (C3)) SNIP::POS P2 M2! DEFAULT-DEFAULTCT))
    *KNOWN-INSTANCES* =
                      ((((P1 . M2!) (V1 . B) (V2 . C)) (HYP (C3)) SNIP::POS)
                       (((P1 . M1!) (V1 . A) (V2 . B)) (HYP (C2)) SNIP::POS))
    *NODE* = P1
    *NAME* = SNIP::NON-RULE

>>>>>> Leaving process id = p7 with bindings:

    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
    *OUTGOING-CHANNELS* =
                        ((NIL NIL DEFAULT-DEFAULTCT M3! OPEN)
                         (NIL ((V3 . V2) (V2 . V1)) DEFAULT-DEFAULTCT P2 OPEN)
                         (NIL ((V4 . V1) (V5 . V2)) DEFAULT-DEFAULTCT P4 OPEN))
    *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-
DEFAULTCT V3 OPEN)
                         ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1!
OPEN))
    *REQUESTS* = NIL
    *REPORTS* = NIL
    *KNOWN-INSTANCES* = ((((P1 . M2!) (V1 . B) (V2 . C)) (DER (C3)) SNIP::POS)
                        (((P1 . M1!) (V1 . A) (V2 . B)) (DER (C2)) SNIP::POS))
    *NODE* = P1
    *NAME* = SNIP::NON-RULE
```

```
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p10| MULTI::|p8| MULTI::|p1|), L: ()

>>>>>> Entering process id = p10 with bindings:

    *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT
    *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT
    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
    *INTRODUCTION-CHANNELS* = NIL
    *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1)
                          ((P2 P1) ((P2 P1))
                           (((P2 P1)
                             (((P1 . M1!) (V1 . A) (P2 . M2!) (V3 . C) (V2 . B)) 2 0
                              ((P2 (DER (C3)) SNIP::TRUE) (P1 (DER (C2)) SNIP::TRUE)) NIL))
                            (P2 (((P2 . M1!) (V3 . B) (V2 . A)) 1 0 ((P2 (DER (C2)) SNIP::TRUE)) NIL)
                             (((P2 . M2!) (V3 . C) (V2 . B)) 1 0 ((P2 (DER (C3)) SNIP::TRUE)) NIL))
                            (P1 (((P1 . M1!) (V1 . A) (V2 . B)) 1 0 ((P1 (DER (C2)) SNIP::TRUE)) NIL)
                             (((P1 . M2!) (V1 . B) (V2 . C)) 1 0 ((P1 (DER (C3)) SNIP::TRUE)) NIL))))))
    *OUTGOING-CHANNELS* = NIL
    *INCOMING-CHANNELS* =
           (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN))
    *REQUESTS* = NIL
    *REPORTS* =
              ((((P1 . M1!) (V1 . A) (V2 . B)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT)
               (((P1 . M2!) (V1 . B) (V2 . C)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT))
    *KNOWN-INSTANCES* = NIL
    *NODE* = M3!
    *TYPE* = SNIP::AND-ENTAILMENT
    *NAME* = SNIP::RULE

>>>>>> Leaving process id = p10 with bindings:

    *USABILITY-TEST* = SNIP::USABILITY-TEST.&-ENT
    *RULE-HANDLER* = SNIP::RULE-HANDLER.&-ENT
    *PRIORITY* = SNIP:HIGH
    *PENDING-FORWARD-INFERENCES* = NIL
```

INTRODUCTION-CHANNELS = NIL *RULE-USE-CHANNELS* = (((NIL NIL DEFAULT-DEFAULTCT P3 OPEN) (P2 P1) ((P2 P1) ((P2 P1)) (((P2 P1) (((P1 . M1!) (V1 . A) (P2 . M2!) (V3 . C) (V2 . B)) 2 0 ((P2 (DER (C3)) SNIP::TRUE) (P1 (DER (C2)) SNIP::TRUE)) NIL)) (P2 (((P2 . M1!) (V3 . B) (V2 . A)) 1 0 ((P2 (DER (C2)) SNIP::TRUE)) NIL) (((P2 . M2!) (V3 . C) (V2 . B)) 1 0 ((P2 (DER (C3)) SNIP::TRUE)) NIL)) (P1 (((P1 . M1!) (V1 . A) (V2 . B)) 1 0 ((P1 (DER (C2)) SNIP::TRUE)) T) (((P1 . M2!) (V1 . B) (V2 . C)) 1 0 ((P1 (DER (C3)) SNIP::TRUE)) T)))))) *OUTGOING-CHANNELS* = NIL *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = NIL *NODE* = M3! *TYPE* = SNIP::AND-ENTAILMENT *NAME* = SNIP::RULE	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p8| MULTI::|p1|), L: () >>>>>> Entering process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN) (NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT)) *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS)) *NODE* = P2 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p8 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL ((V1 . V2) (V2 . V3)) DEFAULT-DEFAULTCT P1 OPEN) (NIL NIL DEFAULT-DEFAULTCT M3! OPEN) (NIL ((V4 . V2) (V5 . V3)) DEFAULT-DEFAULTCT P4 OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P2 . M1!) (V3 . B) (V2 . A)) (DER (C2)) SNIP::POS) (((P2 . M2!) (V3 . C) (V2 . B)) (DER (C3)) SNIP::POS)) *NODE* = P2 *NAME* = SNIP::NON-RULE	
>>>>>> MULTI QUEUES> A: (), H: (MULTI::|p1|), L: () >>>>>> Entering process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = ((((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS P1 M1! DEFAULT-DEFAULTCT) (((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS P1 M2! DEFAULT-DEFAULTCT))	Some residual reports received by

KNOWN-INSTANCES = ((((P4 . M4!) (V4 . A) (V5 . C)) (DER (C6)) SNIP::POS) (((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE >>>>>> Leaving process id = p1 with bindings: *PRIORITY* = SNIP:HIGH *PENDING-FORWARD-INFERENCES* = NIL *OUTGOING-CHANNELS* = ((NIL NIL DEFAULT-DEFAULTCT SNIP:USER OPEN)) *INCOMING-CHANNELS* = (((NIL) DEFAULT-DEFAULTCT P3 OPEN) ((NIL) DEFAULT-DEFAULTCT P2 OPEN) ((NIL) DEFAULT-DEFAULTCT P1 OPEN) ((NIL) DEFAULT-DEFAULTCT V3 OPEN) ((NIL) DEFAULT-DEFAULTCT V2 OPEN) ((NIL) DEFAULT-DEFAULTCT V1 OPEN) ((NIL) DEFAULT-DEFAULTCT M2! OPEN) ((NIL) DEFAULT-DEFAULTCT M1! OPEN)) *REQUESTS* = NIL *REPORTS* = NIL *KNOWN-INSTANCES* = ((((P4 . M4!) (V4 . A) (V5 . C)) (DER (C6)) SNIP::POS) (((P4 . M2!) (V4 . B) (V5 . C)) (DER (C3)) SNIP::POS) (((P4 . M1!) (V4 . A) (V5 . B)) (DER (C2)) SNIP::POS)) *NODE* = P4 *NAME* = SNIP::NON-RULE	P4, which are not propagated to the USER node as they were already known by P4.
>>>>>> MULTI QUEUES> A: (), H: (), L: () **(M4! M2! M1!)** CPU time : 0.72 * End of file C:\Documents and Settings\Pedro Neves\My Documents\TFC\exemplos sneps\ancestors.txt	No more processes queued. The inference process is over! M1!, M2! and M4! are asserted.

Table 12 – The entire deduction process for the commands of Table 11.

9.2. Example: "Married Persons Live Together!"

On this appendix we give SNePSLOG rules and facts for an example that helps to clarify the impact of the work described on chapter 5, "Controlling antecedents". An analysis of the results obtained and some numbers related with the number of processes executed are given on section 5.5.1.

In this example we are interested in answering the query "Who lives with Sofia?", knowing that all married man and woman live together. The SNePSLOG input file is shown on Table 4 and the SNePSLOG command used to trigger the inference process is

ask liveTogether(?who, Sofia).

9.2.1. Deduction using the idle-queue

Table 13 shows the inference produced when we ask who lives together with Sofia using the idle queue.

I wonder if
((P5 (A1 V3) (A2 (SOFIA)) (R (LIVETOGETHER))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P2 (A1 (V2 <-- SOFIA)) (R (WOMAN))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M6! (A1 (SOFIA)) (R (WOMAN))))

"Current antecedent processes working:"
(P2 P3)

I wonder if
((P3 (A1 V1) (A2 (V2 <-- SOFIA)) (R (MARRIED))))
holds within the BS defined by context DEFAULT-DEFAULTCT

CPU time : 0.00

Table 13 – Inference produced for the question "Who lives together with Sofia?", using the idle queue.

9.2.2. Deduction without using the idle-queue

For the same question, the inference produced when the idle queue is not used is given on Table 14. The red color was used to emphasize the inference steps that were avoided when the idle queue was used.

I wonder if
((P5 (A1 V3) (A2 (SOFIA)) (R (LIVETOGETHER))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P3 (A1 V1) (A2 (V2 <-- SOFIA)) (R (MARRIED))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P2 (A1 (V2 <-- SOFIA)) (R (WOMAN))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P1 (A1 V1) (R (MAN))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M6! (A1 (SOFIA)) (R (WOMAN))))

I know
((M7! (A1 (PEDRO)) (R (MAN))))

I know
((M8! (A1 (GUSTAVO)) (R (MAN))))

I know
((M9! (A1 (EMANUEL)) (R (MAN))))

I know
((M10! (A1 (JOSE)) (R (MAN))))

I know
((M11! (A1 (AFONSO)) (R (MAN))))

I know
((M12! (A1 (JOAO)) (R (MAN))))

CPU time : 0.03

Table 14 – Same inference, this time without using the idle-queue.

9.3. Example: "Does John work as a Journalist?"

This example, which has the same purpose as the previous, is a little more complex and will similarly be used to test the work of section 5. This time we are interested in knowing whether John works or not as a journalist knowing that any person has between two and three of the following jobs: Nurse, Policeman, Journalist,

Teacher, Investor, Photographer and Thief. Although we can prove that John is a photographer, it is not possible to apply the and-or inference rule because there is not enough premises for that.

The SNePSLog input file with the rules and facts for this example is presented on Table 6 and the SNePSLOG command used to start inference is

ask Job(John, Journalist).

9.3.1. Deduction using the idle-queue

The inference produced when the idle queue is used is given on Table 15.

```
I wonder if
((M18 (A1 (JOHN)) (A2 (JOURNALIST)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P7 (A1 (V1 <-- JOHN)) (A2 (THIEF)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P6 (A1 (V1 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P5 (A1 (V1 <-- JOHN)) (A2 (INVESTOR)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P4 (A1 (V1 <-- JOHN)) (A2 (TEACHER)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P19 (A1 (V5 <-- JOHN)) (A2 (GREATADVENTURES)) (R
(LIVED))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P18 (A1 (V5 <-- JOHN)) (A2 (DOG)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P16 (A1 (V4 <-- JOHN)) (A2 (LOTSOFMONEY)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M17! (A1 (JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))

"Current antecedent processes working:"
 (P10 P9)

I wonder if
((P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R (IS))))
holds within the BS defined by context DEFAULT-DEFAULTCT
```

```
I know
((M16! (A1 (JOHN)) (A2 (ARTISTIC)) (R (IS))))

"Current antecedent processes working:"
(P10 P9 P8)

I wonder if
((P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R
(HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M15! (A1 (JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R (HAS))))

Since
((M2! (FORALL (V2 <-- JOHN))
 (&ANT (P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R
(WORKINGPLACE))) (P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R
(IS)))
  (P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R
(HAS))))
 (CQ (P11 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))))
and
((P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))
and
((P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R (IS))))
and
((P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R
(HAS))))
I infer
((P11 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))

"Current antecedent processes working:"
 (P10 P9 P8)

"Current antecedent processes working:"
(P7 P6 P5 P4 P2)

I wonder if
((P2 (A1 (V1 <-- JOHN)) (A2 (POLICEMAN)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT


 CPU time : 0.06
```

Table 15 – Deduction process for the "Does John work as a journalist?" example, using the idle queue.

9.3.2. Deduction without using the idle-queue

Similarly, when the idle queue is not used, Table 16 shows that the inference process is slightly longer. Red colored steps are avoided when the idle queue is used.

I wonder if
((M18 (A1 (JOHN)) (A2 (JOURNALIST)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P7 (A1 (V1 <-- JOHN)) (A2 (THIEF)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P6 (A1 (V1 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P5 (A1 (V1 <-- JOHN)) (A2 (INVESTOR)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P4 (A1 (V1 <-- JOHN)) (A2 (TEACHER)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P2 (A1 (V1 <-- JOHN)) (A2 (POLICEMAN)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P1 (A1 (V1 <-- JOHN)) (A2 (NURSE)) (R (JOB))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P19 (A1 (V5 <-- JOHN)) (A2 (GREATADVENTURES)) (R (LIVED))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P18 (A1 (V5 <-- JOHN)) (A2 (DOG)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R (IS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P16 (A1 (V4 <-- JOHN)) (A2 (LOTSOFMONEY)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P14 (A1 (V3 <-- JOHN)) (A2 (HOSPITAL)) (R (WORKINGPLACE))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P13 (A1 (V3 <-- JOHN)) (A2 (NURSING)) (R (STUDIED))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I wonder if
((P12 (A1 (V3 <-- JOHN)) (A2 (SERINGUE)) (R (HAS))))
holds within the BS defined by context DEFAULT-DEFAULTCT

I know
((M17! (A1 (JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))

I know
((M16! (A1 (JOHN)) (A2 (ARTISTIC)) (R (IS))))

I know
((M15! (A1 (JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R (HAS))))

Since
((M2! (FORALL (V2 <-- JOHN))
(&ANT (P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))) (P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R (IS)))
(P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R (HAS))))
(CQ (P11 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))))
and
((P10 (A1 (V2 <-- JOHN)) (A2 (STUDIO)) (R (WORKINGPLACE))))
and
((P9 (A1 (V2 <-- JOHN)) (A2 (ARTISTIC)) (R (IS))))
and
((P8 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHICMACHINE)) (R (HAS))))
I infer
((P11 (A1 (V2 <-- JOHN)) (A2 (PHOTOGRAPHER)) (R (JOB))))

CPU time : 0.02

Table 16 – Deduction process for the same example, this time not using the idle queue.

9.4. Example: ”Which Press publishes the Tour De France?”

The example presented here was used on section 6. Its objective is to show the impact of using different search strategies to find solution to the query

ask publishes(?which, TourDeFrance).

On the example, there are several magazines and newspapers specialized on different kinds of news. There are information and events about politics, music, war and sports. The *Portuguese elections*, the *Rolling Stones concert*, *the War in Iraque* and the *PGA Tour* are all events that may be published on press specialized on one of the topics. We are interested in knowing which magazine or newspaper will publish the *Tour de France*.

On previous sections, we presented the entire deductions produced by SNePS. We do not do that here due to the sizes of the deductions and because we want to test multiple search strategies. However section 6.4 presents tables with the number of processes executed during the deduction processes.

9.4.1. The SNePSLOG input file

```
/*Information about the Portuquese Press*/
Newspaper(Sol).
Newspaper(ABola).
Newspaper(24Horas).
Magazine(Visao).
Magazine(AutoHoje).
Magazine(Caras).
Magazine(Blitz).
Type(Sol, PoliticalPress).
Type(ABola, SportsPress).
Type(24Horas, SensationalPress).
Type(Visao, ContentsPress).
Type(AutoHoje, CarPress).
Type(Caras, SocialPress).
Type(Blitz, MusicPress).
News(PortugueseElections).
News(WarInIraque).
News(TourDeFrance).
News(RolingStonesConcert).
all(x) ({Magazine(x), Newspaper(x)} v=> {PortuguesePress(x)}).
all(x,y,z,w) ({News(x), PortuguesePress(y), Issue(x, z), Type(y, z)} \&=> {Publishes(y,x)}).

/*Information about Sports*/
all(x) ({ImportantEvent(x), SportsEvent(x)} \&=> {News(x), Issue(x, SportsPress)}).
all(x,y) ({Competition(x,y), Sports(y)} \&=> {SportsEvent(x)}).
all(x,y) ({Competition(x,y)} => {ImportantEvent(x)}).
Sports(Cicling).
Sports(Football).
Sports(Golf).
Uses(Ciclying, Bicicle).
Uses(Football, Ball).
Competition(PGATour, Golf).
Competition(TourDeFrance, Cicling).
Competition(UEFACup, Football).
Competition(ChampionsLeague, Football).
Competition(RolandGarros, Ciclying).
```

```
/*Information about Politics */
State(Portugal).
Election(PortugueseElections).
Regime(Portugal, Democracy).
all(x,y,z) ({State(x), Citizen(y,x), Regime(x, Democracy)} \&=> {Votes(y,ElectionsOf(x))}).
all(x) ({State(x), Regime(x, Democracy)} \&=> {Election(ElectionsOf(x))}).
all(x) ({Election(x), Candidate(WinnerOfElection(x),x)} \&=> {Winner(WinnerOfElection(x))}).
all(x) (Election(x) => {News(x), Issue(x, PoliticalPress)}).

/*Information about War*/
Divergent(Iraque, USA).
Country(Iraque).
Country(USA).
War(WarInIraque).
MilitaryConflict(USA, Iraque, WarInIraque).
all(x,y) ({Divergent(x,y)} => {Divergent(y,x)}).
all(x,y,z) ({Country(x), Country(y), MilitaryConflict(x,y,z)} \&=> {War(z), Divergent(x,y)}).
all(x) ({War(x)} => {News(x), Issue(x, PoliticalPress)}).

/*Information about Music*/
Popular(RolingStones).
Band(RolingStones).
Band(ScissorSisters).
Band(U2).
Performer(RolingStonesConcert, RolingStones).
Performer(ScissorSistersConcert, ScissorSisters).
Concert(ScissorSistersConcert).
Concert(RolingStonesConcert).
all(x,y) ({Concert(x), Band(y), Performer(x,y), Popular(y)} \&=> {News(x), Issue(x, MusicPress)}).
```

Table 17 – "Which Press publishes the Tour de France?" example