TR-356

Nonmonotonic Parallel Inheritance Network

by

C. Sakama & A. Okumura

March, 1988

Nonmonotonic Parallel Inheritance Network

Chiaki Sakama and *Akira Okumura*

Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F, 1-4-28, Mita, Minato-ku
Tokyo 108, Japan
csnet: sakama%icot.jp@relay.cs.net
uucp:{enea,inria,kddlab,mit-eddie,ukc}!icot!sakama

Abstract

This paper discusses a formalization of nonmonotonic inheritance reasoning in semantic network and presents a parallel inheritance algorithm based on this approach.

1 Background

First, consider such an inheritance hierarchy in semantic network.

Elephants are gray.

African elephants are elephants.

Clyde is an African elephant.

This hierarchy is represented by a set of first order formulae as follows.

$$W = \{\forall x Elephant(x) \supset Gray(x),$$
$$\forall x AfricanElephant(x) \supset Elephant(x),$$
$$AfricanElephant(clyde)\}$$

In this case, $Gray(clyde)$ is deducible from W. That is, inheritance is realized by the repeated application of modus ponens.

However, when there are some exceptions in the hierarchy, the case becomes somewhat complicated, that is, nonmonotonicity can arise to inheritance. Consider the following hierarchy.

Elephants are normally gray.

Royal elephants are elephants, but are not gray.

Clyde is a royal elephant.

This hierarchy can be represented by a set of first order formulae as follows.

$$
\begin{aligned}
W = \{ & \forall x RoyalElephant(x) \supset Elephant(x),\\
& \forall x RoyalElephant(x) \supset \neg Gray(x),\\
& \forall x Elephant(x) \wedge \neg RoyalElephant(x) \supset Gray(x),\\
& RoyalElephant(clyde)\}
\end{aligned}
$$

Such an inheritance hierarchy with exceptions is called a *nonmonotonic inheritance hierarchy*. Suppose that $Elephant(taro)$ is added to W. When *taro* is not a royal elephant, $\neg RoyalElephant(taro)$ must be represented explicitly in W to deduce $Gray(taro)$. Thus, to represent a nonmonotonic inheritance hierarchy by first order logic, it must be represented explicitly with the formulae.

[Etherington83,Etherington87a] formalized such a nonmonotonic inheritance hierarchy by *default logic* [Reiter80]. For example, the above case is represented as:

$$
\begin{aligned}
W &= \{\forall x RoyalElephant(x) \supset Elephant(x),\\
&\quad\ RoyalElephant(clyde)\}
\end{aligned}
$$

$$
D = \{\frac{Elephant(x) : Gray(x) \wedge \neg RoyalElephant(x)}{Gray(x)}\}
$$

Here, D is a set of defaults, and is informally interpreted as: "When $Elephant(x)$ holds, and $Gray(x) \wedge \neg RoyalElephant(x)$ is consistent with this, then infer $Gray(x)$."

As a result, $Elephant(clyde)$ is deduced from first order formulae W, but $RoyalElephant(clyde)$ in W blocks the derivation of $Gray(clyde)$ from default D. Besides, when $Elephant(taro)$ is added to W and *taro* is not a royal elephant, $\neg RoyalElephant(taro)$ need not be represented explicitly in W to deduce $Gray(taro)$. That is, a nonmonotonic inheritance hierarchy can be treated implicitly by default reasoning.

As is pointed out by Touretzky, however, this approach is somewhat impractical because it needs as many inheritance rules as exceptions in the hierarchies. What is worse, update of such a hierarchy requires modification of all the affected defaults as well as the corresponding first order formulae. This becomes increasingly complex as the network grows, and does not make the most of default reasoning.

This paper presents the formalization of nonmonotonic inheritance reasoning by default logic different from Etherington's approach, and also gives a parallel algorithm based on it.

2 Theory of Nonmonotonic Inheritance Reasoning

2.1 Nonmonotonic Inheritance Network

First, a binary relation IS_A is defined as an acyclic relation between an individual and a class, or a subclass and a superclass.

Definition 2.1 Suppose a class (or an individual) x, y, and a set, $Upper_i(x)$ $(i \geq 0)$:

1. $IS\_A(x,y)$ *iff* $x \in y$, *or* $x \subseteq y$.
2. $x \in Upper_0(x)$ *iff* $IS\_A(x,y)$.
3. $z \in Upper_{i+1}(x)$ *iff* $y \in Upper_i(x)$ *and* $IS\_A(y,z)$.
4. *If* $IS\_A(x,y)$, *then* $x \notin \bigcup_{i \geq 1} Upper_i(x)$. □

The fourth condition above denotes that the IS_A hierarchy is *acyclic*. Next, a nonmonotonic inheritance network is defined.

Definition 2.2 A *nonmonotonic inheritance network* $\Delta = (W, D)$ is defined as follows.

W : *a consistent set of ground instances of either* $IS\_A(x,y)$, $Property(z,w)$, *or* $\neg Property(u,v)$.

$$D = \{\frac{IS\_A(x,y) \land Property(y,z) : Property(x,z)}{Property(x,z)}, \frac{IS\_A(x,y) \land \neg Property(y,z) : \neg Property(x,z)}{\neg Property(x,z)}\} \quad \square$$

In the above definition, $Property(y,z)$ (or $\neg Property(y,z)$) denotes that a class or an individual y has (or has not) a property z.

Example 2.1 Suppose the following well-known nonmonotonic inheritance hierarchies.

Molluscs are normally shellbearers.

Cephalopods are molluscs but are not normally shellbearers.

Nautili are cephalopods but are shellbearers.

Fred is a nautilus.

In the above hierarchy, cephalopods become an exception to molluscs with respect to the property of shellbearers, and nautili also become an exception to cephalopods with respect to the property of shellbearers.

Such a hierarchy is represented by $\Delta = (W, D)$, where

$$\begin{aligned} W = \{ & IS\_A(cephalopod, mollusc), \\ & IS\_A(nautilus, cephalopod), \\ & IS\_A(fred, nautilus), \\ & Property(mollusc, has\_shell), \\ & \neg Property(cephalopod, has\_shell), \\ & Property(nautilus, has\_shell)\} \end{aligned}$$

As a result, the extension of Δ becomes:

$E = W \cup \{Property(fred, has\_shell)\}$. (Informally, an extension denotes a set of logical consequences of a default theory.) □

The above example is represented by [Etherington83,Etherington87a], as follows.[1]

$$\begin{aligned} W \quad = \quad & \{\forall x Cephalopods(x) \supset Mollusc(x), \\ & \forall x Nautilus(x) \supset Cephalopods(x), \\ & \forall x Nautilus(x) \supset Shellbearer(x), \\ & Nautilus(fred)\} \end{aligned}$$

$$\begin{aligned} D \quad = \quad & \{\frac{Mollusc(x) : Shellbearer(x) \wedge \neg Cephalopods(x)}{Shellbearer(x)}, \\ & \frac{Cephalopods(x) : \neg Shellbearer(x) \wedge \neg Nautilus(x)}{\neg Shellbearer(x)}\} \end{aligned}$$

Compared with Etherington's formalization, the approach presented here can give a simple semantics for the interpretation of nonmonotonic inheritance reasoning. This is because the nonmonotonicity in the inheritance of property is treated separately from the IS_A hierarchy between classes.

In this approach, the data in the network W can be described separately from its inheritance rules D which are given as a general rule for inheritance reasoning. This leads to a simple

[1][Etherington87b] employs a different manner of representation, based on Touretzky's approach.

description of a network, and furthermore, update of the network can be achieved only through modification of the corresponding data in W and there is no need to modify defaults.

Note that the transitivity does not hold for the IS_A relation in Δ, then $IS\_A(fred, cephalopod)$, for example, cannot be derived in the above example. If it is required, however, by representing it as a property such as $Property(nautilus, upper(cephalopod))$, $Property(fred, upper(cephalopod))$ can be derived.

Δ is called a *normal default theory* and has at least one consistent extension for every consistent W [Reiter80].

Definition 2.3 A nonmonotonic inheritance network Δ is ***definite*** iff it has only one extension. □

Example 2.1 is a definite case. However, there is an indefinite network which has multiple extensions being inconsistent with each other. For instance, Example 2.1 becomes indefinite if $IS\_A(fred, cephalopod)$ is added to W. It generates two extensions; Fred has shell in one extension, while he does not have it in the other. Such an indefinite case is discussed in the next.

2.2 Nixon Diamond

When multiple inheritance is considered in a nonmonotonic inheritance network, there is sometimes a problem for ambiguity. Consider the well-known *Nixon diamond* problem. That is, Nixon is both a Quaker and a Republican, and Quakers are typically pacifists, while Republicans typically are not. Then, the problem is whether Nixon is a pacifist or not.

In Etherington's manner, it is represented as follows.

$$\begin{aligned} W &= \{\forall x Nixon(x) \supset Quaker(x), \\ &\quad \forall x Nixon(x) \supset Republican(x), \\ &\quad Nixon(nixon)\} \end{aligned}$$

$$D = \{\frac{Quaker(x) : Pacifist(x)}{Pacifist(x)}, \frac{Republican(x) : \neg Pacifist(x)}{\neg Pacifist(x)}\}$$

As a result, there exists the following two extensions which are inconsistent with each other.

$$E_1 = \{Nixon(nixon), Quaker(nixon), Republican(nixon), Pacifist(nixon)\}$$
$$E_2 = \{Nixon(nixon), Quaker(nixon), Republican(nixon), \neg Pacifist(nixon)\}$$

The same situation happens in $\Delta = (W, D)$ as:

$$E_1 = W \cup \{Property(nixon, pacifist)\}$$
$$E_2 = W \cup \{\neg Property(nixon, pacifist)\}$$

where

$$\begin{aligned} W = \{&IS\_A(nixon, quaker), \\ &IS\_A(nixon, republican), \\ &Property(quaker, pacifist), \\ &\neg Property(republican, pacifist)\}. \end{aligned}$$

Such a network which has more than one extension is called *indefinite*, and there are two attitudes for treating such an indefinite network.

A *skeptical reasoner* draws no conclusion from ambiguous information, and hence offers no conclusion as to whether Nixon is a pacifist or not. A *credulous reasoner*, on the other hand, tries to draw as many conclusions as possible, and hence offers two alternatives: Nixon is a pacifist in one case, and is not a pacifist in the other case [Touretzky87].

The skeptical attitude is superior to the credulous one from an algorithmic point of view. Since a skeptical reasoner always generates a unique extension, its algorithm is simpler and more efficient than that of the credulous reasoner, which must generate multiple possible extensions that grow exponentially as ambiguity increases. The credulous attitude, however, is considered more expressive than the skeptical attitude since it can represent ambiguous information.

Our algorithm, which is shown in the next section, is close to the credulous attitude in the sense of producing ambiguous information, but it does not generate multiple extensions. That is, it produces a set of properties for an input class, and ambiguous information can be represented as one of its properties.

3 Parallel Inheritance

3.1 π Algorithm

Inheritance algorithms combined with parallelism have been studied over the past few years. NETL [Fahlman79] is well-known as a semantic network system. In NETL, inheritance is performed by *parallel marker propagation* over nodes in a network. As is pointed out by Etherington, however, NETL does not treat nonmonotonic cases correctly.

[Touretzky86] have proposed some inheritance algorithms for a nonmonotonic inheritance system. Those algorithms are based on the choice of inference paths in multiple inheritance, and limited parallelism is achieved. They offer creduious inference system and also skeptical version is discussed in [Horty87]. However, they require each derived path to contain its entire derivation history and it becomes overloaded as the size of the network increases.

[Etherington83,Etherington87a] have shown a parallel algorithm based on his formalization and proved its correctness, that is, all inferences lie within a single extension. However, his algorithm is not complete in general: there are some extensions which do not come out from the algorithm. [Cottrell85] has also shown a parallel *connectionist* model based on Etherington's approach, but there is no assurance of correctness and the example given there is quite simple.

Now we show a π *algorithm* (parallel inheritance algorithm) for the nonmonotonic inheritance network presented in the previous section. The notation in the algorithm corresponding to Δ is as follows.

$property(class, CProps)$ where $CProps \neq \emptyset$ *iff* $\forall cprop \in CProps$,
$\exists Property(class, cprop) \in W$ or $\forall not(cprop) \in CProps$, $\exists \neg Property(class, cprop) \in W$.
$property(class, \emptyset)$ *iff* $\forall cprop$, $Property(class, cprop) \notin W$ and $\neg Property(class, cprop) \notin W$.
$is\_a(class, Uppers)$ where $Uppers \neq \emptyset$ *iff* $\forall upper \in Uppers$, $\exists IS\_A(class, upper) \in W$.
$is\_a(class, \emptyset)$ *iff* $\forall upper$, $IS\_A(class, upper) \notin W$.

Notation which begins with a capital letter in the following procedures denotes a set.

```
procedure $\pi(input : class, output : Props)$;
  begin
    $property(class, CProps)$;
    $is\_a(class, Uppers)$;
    $Temp \leftarrow \emptyset$;
    while $Uppers \neq \emptyset$ do
      begin
        select $upper$ from $Uppers$;
        call $\pi(upper, UProps)$;
        $Temp \leftarrow Temp \cup UProps$;
        $Uppers \leftarrow Uppers - \{upper\}$
      end
    call $reverse(CProps, RevCProps)$;
    $Props \leftarrow CProps \cup (Temp - RevCProps)$
   end

procedure reverse(input : CProps, output : RevCProps);
   begin
    if $CProps = \emptyset$ then $RevCProps = \emptyset$
    else
    while $CProps \neq \emptyset$ do
     begin
      select cprop from CProps;
      if $cprop = not(prop)$ then $prop \in RevCProps$
      else $not(cprop) \in RevCProps$
     end
   end
```

Procedure π produces a set of properties for an input class, and procedure *reverse* turns over the properties as: $prop \leftrightarrow not(prop)$.

Example 3.1 To see how this procedure works, we apply this algorithm to Example 2.1.

First, the following is defined from W.

$property(mollusc, \{has\_shell\})$.
$property(cephalopod, \{not(has\_shell)\})$.
$property(nautilus, \{has\_shell\})$.
$property(fred, \emptyset)$.
$is\_a(mollusc, \emptyset)$.
$is\_a(cephalopod, \{mollusc\})$.
$is\_a(nautilus, \{cephalopod\})$.
$is\_a(fred, \{nautilus\})$.

Then $\pi(fred, Props)$ works as follows.

```
$property(fred, \emptyset)$;
$is\_a(fred, \{nautilus\})$;
$Temp1 \leftarrow \emptyset$;
call $\pi(nautilus, UProps1)$;
```

```
    $property(nautilus, \{has\_shell\});$
    $is\_a(nautilus, \{cephalopod\});$
    $Temp2 \leftarrow \emptyset;$
    $call\ \pi(cephalopod, UProps2);$
        $property(cephalopod, \{not(has\_shell)\});$
        $is\_a(cephalopod, \{mollusc\});$
        $Temp3 \leftarrow \emptyset;$
        $call\ \pi(mollusc, UProps3);$
            $property(mollusc, \{has\_shell\});$
            $is\_a(mollusc, \emptyset);$
            $Temp4 \leftarrow \emptyset;$
            $Uppers4 = \emptyset$
            $call\ reverse(\{has\_shell\}, \{not(has\_shell)\};$
            $UProps3 \leftarrow \{has\_shell\} \cup (\emptyset - \{not(has\_shell)\})$
                        $= \{has\_shell\}$
        $\pi(mollusc, \{has\_shell\});$
        $Temp3 \leftarrow \emptyset \cup \{has\_shell\};$
        $Uppers3 \leftarrow \{mollusc\} - \{mollusc\}$
                  $= \emptyset;$
        $call\ reverse(\{not(has\_shell)\}, \{has\_shell\});$
        $UProps2 \leftarrow \{not(has\_shell)\} \cup (\{has\_shell\} - \{has\_shell\})$
                    $= \{not(has\_shell)\}$
    $\pi(cephalopod, \{not(has\_shell)\});$
    $Temp2 \leftarrow \emptyset \cup \{not(has\_shell)\};$
    $Uppers2 \leftarrow \{cephalopod\} - \{cephalopod\}$
              $= \emptyset;$
    $call\ reverse(\{has\_shell\}, \{not(has\_shell)\});$
    $UProps1 \leftarrow \{has\_shell\} \cup (\{not(has\_shell)\} - \{not(has\_shell)\})$
                $= \{has\_shell\}$
$\pi(nautilus, \{has\_shell\});$
$Temp1 \leftarrow \emptyset \cup \{has\_shell\};$
$Uppers1 \leftarrow \{nautilus\} - \{nautilus\}$
```

$= \emptyset;$

call reverse$(\emptyset, \emptyset)$;

$Props \leftarrow \emptyset \cup (\{has\_shell\} - \emptyset)$

$= \{has\_shell\}$

$\pi(fred, \{has\_shell\})$. $\square$

In the π procedure, multiple inheritance is achieved when there is more than one upper class for a class in $is\_a(class, Uppers)$. Then each upper class can independently call the π procedure recursively, parallel execution is achieved in multiple inheritance.

3.2 Δ vs. π

Here, the soundness and completeness of the π procedure with respect to a nonmonotonic inheritance network Δ is shown.

Proposition Suppose a nonmonotonic inheritance network Δ and its extension E, then

$\forall class, \pi(class, Props)$ *iff*

$$Props = \{prop \mid \exists E, Property(class, prop) \in E\}$$
$$\cup\{not(prop) \mid \exists E, \neg Property(class, prop) \in E\}.$$

Proof Suppose first that $\exists n, n+1, Upper_n(class) \neq \emptyset, Upper_{n+1}(class) = \emptyset$, then $\forall c_n \in Upper_n(class)$, $is\_a(c_n, \emptyset)$ holds. Assume $\pi(c_n, Props_n)$, then

$$Props_n = CProps_n \cup (Temp_n - RevCProps_n)$$
$$= CProps_n \cup (\emptyset - RevCProps_n)$$
$$= CProps_n$$
$$= \{cprop \mid Property(c_n, cprop) \in W\} \cup \{not(cprop) \mid \neg Property(c_n, cprop) \in W\}.$$

Next assume $\forall c_i \in Upper_i(class)$ $(0 < i \leq n)$, $\pi(c_i, Props_i)$ where

$$Props_i = \{prop \mid \exists E, Property(c_i, prop) \in E\} \cup \{not(prop) \mid \exists E, \neg Property(c_i, prop) \in E\}.$$

holds.

Let $\forall c_{i-1} \in Upper_{i-1}(class)$, $\pi(c_{i-1}, Props_{i-1})$, then

$$Props_{i-1} = CProps_{i-1} \cup (Temp_{i-1} - RevCProps_{i-1}).$$

(a) If $uprop \in CProps_{i-1}$ or $not(uprop) \in CProps_{i-1}$, then

$Property(c_{i-1}, uprop) \in W$ or $\neg Property(c_{i-1}, uprop) \in W$.

(b) Otherwise, $uprop \in Temp_{i-1} - RevCProps_{i-1}$ or $not(uprop) \in Temp_{i-1} - RevCProps_{i-1}$, then

$uprop \in Temp_{i-1}$ or $not(uprop) \in Temp_{i-1}$ where $Temp_{i-1} = Props_i$.

By the assumption, $\forall c_i \in Upper_i(class)$,

$\exists E, Property(c_i, uprop) \in E$ or $\exists E, \neg Property(c_i, uprop) \in E$.

In case $uprop \in Temp_{i-1}$, clearly $uprop \notin RevCProps_{i-1}$, then

$\neg Property(c_{i-1}, uprop) \notin W$ and $Property(c_{i-1}, uprop) \in E$.

In case $not(uprop) \in Temp_{i-1}$, clearly $not(uprop) \notin RevCProps_{i-1}$, then

$Property(c_{i-1}, uprop) \notin W$ and $\neg Property(c_{i-1}, uprop) \in E$.

Therefore,

$$Props_{i-1} \subseteq \{prop \mid \exists E, Property(c_{i-1}, prop) \in E\}$$
$$\cup \{not(prop) \mid \exists E, \neg Property(c_{i-1}, prop) \in E\} \quad (*).$$

Let $\forall c_{i-1} \in Upper_{i-1}(class)$, $\exists E$, $Property(c_{i-1}, uprop) \in E$ or $\neg Property(c_{i-1}, uprop) \in E$.

(a) If $Property(c_{i-1}, uprop) \in W$ or $\neg Property(c_{i-1}, uprop) \in W$, then

$uprop \in CProps_{i-1}$ or $not(uprop) \in CProps_{i-1}$.

(b) Otherwise, $\exists E, Property(c_i, uprop) \in E$ or $\neg Property(c_i, uprop) \in E$.

By the assumption, $\forall c_i \in Upper_i(class)$, $uprop \in Props_i$, or $not(uprop) \in Props_i$,

where $\pi(c_i, Props_i)$.

In case $\exists E, Property(c_{i-1}, uprop) \in E$, $\neg Property(c_{i-1}, uprop) \notin W$ holds, then

$uprop \notin RevCProps_{i-1}$ and $uprop \in Props_i - RevCProps_{i-1}$.

Then $uprop \in CProps_{i-1} \cup (Temp_{i-1} - RevCProps_{i-1})$.

In case $\exists E, \neg Property(c_{i-1}, uprop) \in E$, $Property(c_{i-1}, uprop) \notin W$ holds, then

$not(uprop) \notin RevCProps_{i-1}$ and $not(uprop) \in Props_i - RevCProps_{i-1}$.

Then $not(uprop) \in CProps_{i-1} \cup (Temp_{i-1} - RevCProps_{i-1})$.

Hence $uprop \in Props_{i-1}$, or $not(uprop) \in Props_{i-1}$, where $\pi(c_{i-1}, Props_{i-1})$.

Therefore,

$$Props_{i-1} \supseteq \{prop \mid \exists E, Property(c_{i-1}, prop) \in E\}$$
$$\cup \{not(prop) \mid \exists E, \neg Property(c_{i-1}, prop) \in E\} \quad (\dagger).$$

Together from $(*)$ and $(\dagger)$,

$$Props_{i-1} = \{prop \mid \exists E, Property(c_{i-1}, prop) \in E\}$$

$$\cup\{not(prop) \mid \exists E, \neg Property(c_{i-1}, prop) \in E\}.$$

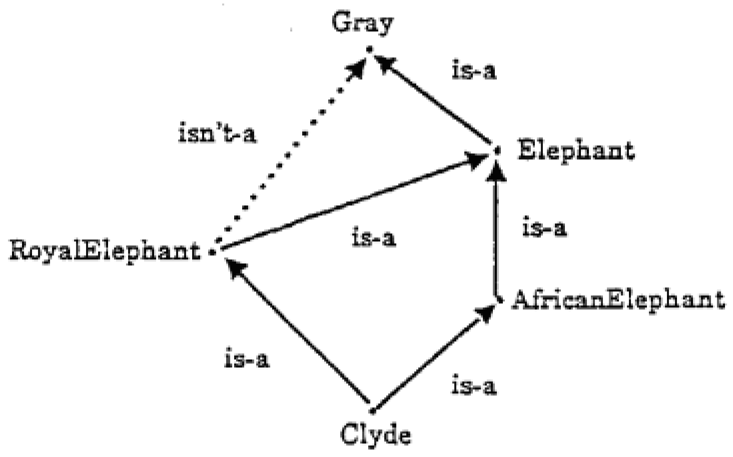
By induction, we have the desired result. $\square$

4 Discussion

The previous sections presented a formalization of the nonmonotonic inheritance network by a normal default theory. It enables us to define inheritance rules apart from data in a network, and improves readability or maintenance of a network. The problem is, as is mentioned in section 2.1, some redundant IS_A link, which is defined transitively, makes a definite network indefinite.

Besides, the parallel inheritance algorithm based on this approach, generates a set of all inheritable properties for an input class. When a network is definite, those properties are included in an extension. While in a case of indefinite network, it collects ambiguous information from multiple extensions, and then the result of the algorithm is *not correct* in a sense. However, the requirement for the correctness of the algorithm is apparently to defend resultant extensions from logical inconsistency. And in π algorithm, the output is not the extension itself, but a set of properties for an input class, and the ambiguous information is pushed into argument, so there is no problem logically.

In general, the interpretation of indefinite network is not straightforward; there are many pathological cases. Let us consider an example of Sandewall's type 1c problem. [Sandewall86] has defined some basic structure types for inheritance network and given sound inference rules for these structures. His type 1c structure is as follows.



In this case, Etherington or Touretzky yields two alternatives; Clyde is gray in one case and is not gray in the other, while Sandewall's approach only concludes that Clyde is not gray. The same with Sandewall's result is achieved in [Brewka87], where he formalized nonmonotonic inheritance system in his frame-like language using *variable circumscription* [McCarthy86]. In our algorithm, for an input class Clyde, two alternative properties, gray and not(gray) are generated as output.

There is no unique interpretation for such an indefinite network, and which is to be preferred cannot be judged in general. (*Clyde only knows his color.*)

References

[Brewka87] Brewka,G.: "The Logic of Inheritance in Frame Systems", *IJCAI'87*, pp.483-488, 1987.

[Cottrell85] Cottrell,G.W.: "Parallelism in Inheritance Hierarchies with Exception", *IJCAI'85*, pp.194-202, 1985.

[Etherington83] Etherington,D.W. and Reiter,R.: "On Inheritance Hierarchies with Exceptions", *AAAI'83*, pp.104-108, 1983.

[Etherington87a] Etherington,D.W.: "Formalizing Nonmonotonic Reasoning Systems", *Artificial Intelligence 31*, pp.41-85, 1987.

[Etherington87b] Etherington,D.W.: "More on Inheritance Hierarchies with Exceptions", *AAAI'87*, pp.352-357, 1987.

[Fahlman79] Fahlman,S.E.: "NETL: A System for Representing and Using Real-World Knowledge", *MIT Press*, Cambridge, MA, 1979.

[Horty87] Horty,J.F., Thomason,R.H. and Touretzky,D.S.: "A Skeptical Theory of Inheritance", *AAAI'87*, pp.358-363, 1987.

[McCarthy86] McCarthy,J.:"Applications of Circumscription to Formalizing Common Sense Knowledge", *Artificial Intelligence 28*, pp.89-116, 1986.

[Reiter80] Reiter,R.: "A Logic for Default Reasoning", *Artificial Intelligence 13*, pp.81-132, 1980.

[Sandewall86] Sandewall,E.: "Nonmonotonic Inference Rules for Multiple Inheritance with Exceptions", *Proc. of IEEE*, vol.74, pp.1345-1353, 1986.

[Touretzky86] Touretzky,D.S.: "The Mathematics of Inheritance Systems", *Research Notes in Artificial Intelligence*, Pitman, London, 1986.

[Touretzky87] Touretzky,D.S., Horty,J.F. and Thomason,R.H.: "A Clash of Intuitions", *IJCAI'87*, pp.476-482, 1987.

[Ueda86] Ueda,K.: "Guarded Horn Clauses", *Lecture Notes in Computer Sciences 221*, Springer-Verlag, Berlin, 1986.

Appendix

Here, we show an implementation of the π algorithm in GHC (Guarded Horn Clauses) [Ueda86]. GHC is the parallel logic programming language developed as the kernel language at ICOT.

The syntax of a clause in GHC is in the following form:

$H : -G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n.$

where the part preceding '|' is called a guard, and the part succeeding it is called a body. A clause with an empty guard is a goal clause. The declarative meaning of the clause is the same as Prolog.

The execution of a GHC program proceeds by reducing a given goal clause to the empty clause as follows.

(a) The guard of a clause cannot export any bindings to the caller of that clause.

(b) The body of a clause cannot export any bindings to the guard of that clause before commitment.

(c) When there are several candidate clauses for invocation, the clause whose guard first succeeds is selected for commitment.

Under these conditions, the execution of goal reduction is done in parallel. Now an interpreter is shown using the example of a shellbearer.

```
/*** Nonmonotonic Parallel Inheritance Network in GHC ***/
%%% inheritance procedure %%%
   pi(Class,Props,Tail):- true |
                                 property(Class,Props,Temp),
                                 is_a(Class,Uppers),
                                 has_property(Uppers,UProps,Res),
                              filter(Props,UProps,Res,Temp,Tail).
  has_property([UClass|Rest],UProps,Tail):- true |
                              pi(UClass,UProps,Temp),
                              has_property(Rest,Temp,Tail).
  has_property([], UProps,Tail):- true | UProps=Tail.
  filter([CProp|Rest], In,Tail1,Out,Tail2):- CProp\=not(_) |
                              filter2(not(CProp),In,Tail1,Temp,Tail3),
                              filter(Rest,Temp,Tail3,Out,Tail2).
  filter([not(CProp)|Rest],In,Tail1,Out,Tail2):- true |
                              filter2(CProp,In,Tail1,Temp,Tail3),
                              filter(Rest,Temp,Tail3,Out,Tail2).
  filter(Out, In,Tail1,Out,Tail2):- true |
                              In=Out,Tail1=Tail2.
  filter2(CProp,[P1|P2],Tail1,Temp,Tail2):- P1\=CProp |
                              Temp=[P1|Rest],
                              filter2(CProp,P2,Tail1,Rest,Tail2).
  filter2(CProp,[P1|P2],Tail1,Temp,Tail2):- P1=CProp |
                              filter2(CProp,P2,Tail1,Temp,Tail2).
  filter2(CProp,Tail1, Tail1,Temp,Tail2):- true |
                              Temp=Tail2.

%%% data %%%
  is_a(mollusc, Uppers):- true | Uppers=[].
  is_a(aquatic, Uppers):- true | Uppers=[].
  is_a(cephalopod, Uppers):- true | Uppers=[mollusc,aquatic].
  is_a(nautilus, Uppers):- true | Uppers=[cephalopod].
  is_a(fred, Uppers):- true | Uppers=[nautilus].
  property(mollusc, CProps,Tail):- true | CProps=[soft_body,has_shell|Tail].
  property(aquatic, CProps,Tail):- true | CProps=[swimming|Tail].
  property(cephalopod,CProps,Tail):- true | CProps=[not(has_shell),has_legs|Tail].
  property(nautilus, CProps,Tail):- true | CProps=[has_shell,not(swimming)|Tail].
  property(fred, CProps,Tail):- true | CProps=[american|Tail].
```

```
%%% execution results %%%
   | ?- ghc pi(fred,Props,[]).
   21 msec.
   Props = [american,has_shell,not(swimming),has_legs,soft_body]
   yes
```

This GHC program is easily translated into a Prolog program, which performs sequential inheritance in a network.