Lecture Notes in Artificial Intelligence

Edited by J. Siekmann

Subseries of Lecture Notes in Computer Science

383

K. Furukawa H. Tanaka
T. Fujisaki (Eds.)

Logic Programming '88

Proceedings of the 7th Conference
Tokyo, Japan, April 11–14, 1988

Springer-Verlag

Berlin Heidelberg New York London Paris Tokyo Hong Kong

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

Abstract

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

1 Background

First, consider such an inheritance hierarchy in a 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 exceptions in the hierarchy, the case becomes somewhat more complicated. 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.

$$W = \{\forall x RoyalElephant(x) \supset Elephant(x),$$

$$\forall x RoyalElephant(x) \supset \neg Gray(x),$$
$$\forall x Elephant(x) \wedge \neg RoyalElephant(x) \supset Gray(x),$$
$$RoyalElephant(clyde)\}$$

Suppose we add the fact, $Elephant(taro)$, to W. When $taro$ is not a royal elephant, the fact, $\neg RoyalElephant(taro)$, also has to be represented explicitly in W to deduce $Gray(taro)$. Thus, to represent such an inheritance hierarchy in first order formulae, all the negative information has to be represented explicitly.

[ER83,Eth87a] represented such an inheritance hierarchy by Reiter's *default logic* [Rei80]. For example, the above hierarchy is represented as:

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

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

D is a set of defaults and read 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 W, but $RoyalElephant(clyde)$ in W blocks the derivation of $Gray(clyde)$ from the default D. Besides, when $Elephant(taro)$ is added to W and $taro$ is not a royal elephant, $Gray(taro)$ can be deduced from D without $\neg RoyalElephant(taro)$. Such an inheritance hierarchy is called a *nonmonotonic inheritance hierarchy*, and it enables us to treat exceptions implicitly.

This formulation, however, seems to have some drawbacks since it requires as many inheritance rules as exceptions in a hierarchy. Moreover, update of such a hierarchy requires modification of all the affected default rules as well as the corresponding first order statements. It will become increasingly complex as the network grows, and does not make the most of default reasoning.

The following sections present the formalization of nonmonotonic inheritance reasoning by default logic in a different manner from Etherington, and also give a parallel algorithm based on this approach.

2 Theory of Nonmonotonic Inheritance Reasoning

First, two predicates are introduced.

1. $IS\_A(x, y)$: an acyclic relation between an individual x and a class y, or a subclass x and a superclass y; that is, $IS\_A(x, y)\ iff\ x \in y,\ or\ x \subseteq y$.

2. $Property(x, y)$ (*resp.* $\neg Property(x, y)$): a class or an individual x has (*resp.* has not) a property y.

Now a nonmonotonic inheritance network is defined.

Definition 2.1 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* $\neg Property(u, v)$.

$$D = \{\frac{IS\_A(x, y) \wedge Property(y, z) : Property(x, z)}{Property(x, z)}, \frac{IS\_A(x, y) \wedge \neg Property(y, z) : \neg Property(x, z)}{\neg Property(x, z)}\}$$ □

Example 2.1 Suppose the following classical hierarchy.

Molluscs are normally shellbearers.

Cephalopods are molluscs but are not normally shellbearers.

Nautili are cephalopods but are shellbearers.

Fred is a nautilus.

In this hierarchy, cephalopods become an exception to molluscs with respect to shellbearers, and nautili also become an exception to cephalopods with respect to 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 same hierarchy is represented by [ER83,Eth87a] as:[1]

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

$$
D \;=\; \{\frac{Mollusc(x) : Shellbearer(x) \wedge \neg Cephalopod(x)}{Shellbearer(x)}, \quad \frac{Cephalopod(x) : \neg Shellbearer(x) \wedge \neg Nautilus(x)}{\neg Shellbearer(x)}\}
$$

Compairing these two formulations, our approach describes the data in the network, W, apart from the inheritance rules, D. This provides a simple description of inheritance hierarchies, since default rules are independent of the network. It also facilitates updation of the network, since it requires changing only the corresponding data in W and does not need to modify defaults.

Such a system, which consists of a set of classes and a set of inheritable properties associated with each class, is called a *class/property inheritance system* [Tou86]. Our approach defines exceptions as nonmonotonic properties of classes, while IS_A hierarchy defines a monotonic relation between classes. In the IS_A hierarchy, transitivity is not assumed since it generates a redundant link. For example, $IS\_A(fred, cephalopod)$ is deduced using transitivity in Example 2.1, then it leads to an extension which contains $\neg Property(fred, has\_shell)$, which is an undesired result. To derive a transitive class-subclass relation, it is necesarry to add, for example, $Property(nautilus, upper(cephalopod))$ to W, to derive $Property(fred, upper(cephalopod))$.

Δ is called a *normal default theory* and has at least one consistent extension for every consistent W [Rei80]. Nonmonotonic inheritance networks are classified by the number of their extensions.

Definition 2.2 A nonmonotonic inheritance network Δ is *definite* iff it has only one extension. □

Example 2.1 is a definite case. However, there is a network with multiple extensions which are inconsistent with each other. Consider the notorious example of *Nixon diamond*. The problem is: *Nixon is both a Quaker and a Republican, and Quakers are typically pacifists,*

[1] [Eth87b] employs a different manner of representation, based on Touretzky's approach.

while Republicans are typically not. Then, whether Nixon is a pacifist or not?

This hierarchy is represented in Δ with W:

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

In this case, there exists the following two extensions which are inconsistent with each other.

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

Such a network which has more than one extension is called *indefinite*, and there are two attitudes for treating such an indefinite network [THT87]. 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 in the other.

From an algorithmic point of view, a skeptical reasoner always generates a unique extension, then its algorithm seems to be simpler and more efficient than that of the credulous one, which generates multiple possible extensions that grow exponentially as ambiguity increases. The credulous attitude, on the other hand, seems to be more expressive than the skeptical attitude, since the explicit representation of ambiguities suggests that there is some disagreement in the network structure.

To take advantage of each attitude, an algorithm which can treat ambiguities but does not generate multiple extensions is considered. The next section discusses such an inheritance algorithm and its parallel execution.

3 Parallel Inheritance Algorithm

Inheritance algorithms combined with parallelism have been studied over the past few years. NETL [Fah79] is a pioneering semantic network system. In NETL, inheritance is performed by *parallel marker propagation* over nodes in a network. However, as is pointed out by [ER83,Eth87a], NETL does not treat nonmonotonic cases correctly.

[Tou86] has 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 a credulous inference system and also a skeptical version is discussed in [HTT87]. These methods, however, require each derived path to contain its entire derivation history and seem to become overloaded as the size of the network increases.

[ER83,Eth87a] have shown a parallel algorithm based on their formalization and proved its correctness, that is, all inferences lie within a single extension. However, this algorithm is not complete in general; there might be some extensions which do not come out of the algorithm. Related to this work, [Cot85] has shown a parallel *connectionist* architecture, but there is no assurance of correctness.

The remainder of this section shows a π *(parallel inheritance) algorithm* for the nonmonotonic inheritance network presented in the previous section. First, notation in the algorithm corresponding to Δ is given:

(a) $property(class, \emptyset)$ *iff* $\forall cprop, Property(class, cprop) \notin W$ and $\neg Property(class, cprop) \notin W$.

Otherwise, $property(class, CProps)$ *iff* $\forall cprop \in CProps$, $Property(class, cprop) \in W$, and $\forall not(cprop) \in CProps$, $\neg Property(class, cprop) \in W$.

(b) $is\_a(class, \emptyset)$ *iff* $\forall upper, IS\_A(class, upper) \notin W$.

Otherwise, $is\_a(class, Uppers)$ *iff* $\forall upper \in Uppers$, $IS\_A(class, upper) \in W$.

Here, $\emptyset$ denotes an empty set and notation which begins with a capital letter denotes a set. Now, the π algorithm is presented below.

```
procedure π(input : class, output : Props);
  begin
   call property(class, CProps);
   call is_a(class, Uppers);
   Temp ← ∅;
   while Uppers ≠ ∅ do
    begin
     select upper from Uppers;
     call π(upper, UProps);
     Temp ← Temp ∪ UProps;
```

$Uppers \leftarrow Uppers - \{upper\}$

end

call reverse$(CProps, RevCProps)$;

$Props \leftarrow CProps \cup (Temp - RevCProps)$

end

procedure reverse$(input : CProps, output : RevCProps)$;

begin

$RevCProps = \emptyset$;

while $CProps \neq \emptyset$ *do*

begin

select cprop from $CProps$;

if $cprop = not(prop)$ *then*

$RevCProps \leftarrow RevCProps \cup \{prop\}$

else $RevCProps \leftarrow RevCProps \cup \{not(cprop)\}$;

$CProps \leftarrow CProps - \{cprop\}$

end

end

Example 3.1 In Example 2.1, $\pi(fred, \{has\_shell\})$ where

$property(mollusc, \{has\_shell\})$, $property(cephalopod, \{not(has\_shell)\})$,

$property(nautilus, \{has\_shell\})$, $property(fred, \emptyset)$, $is\_a(mollusc, \emptyset)$,

$is\_a(cephalopod, \{mollusc\})$, $is\_a(nautilus, \{cephalopod\})$, and $is\_a(fred, \{nautilus\})$.

In Nixon Diamond, $\pi(nixon, \{pacifist, not(pacifist)\})$ where

$property(quaker, \{pacifist\})$, $property(republican, \{not(pacifist)\})$, $property(nixon, \emptyset)$,

$is\_a(quaker, \emptyset)$, $is\_a(republican, \emptyset)$, and $is\_a(nixon, \{quaker, republican\})$. □

The procedure π produces a set of inheritable properties for an input class downwards from upper classes to their subclasses. Nonmonotonic inheritance is achieved through overriding higher properties by conflicting lower ones. When there is more than one upper class at $is\_a(class, Uppers)$, each upper class calls the recursive π procedure independently. This process will be executed in parallel on a massively parallel architecture, where each processor will have cost proportional to the length of the inheritance path. The π algorithm is implemented in the parallel logic programming language GHC (Guarded Horn Clauses) [Ued86],

and is shown in Appendix A.

The next proposition assures the soundness and completeness of the π procedure with respect to a nonmonotonic inheritance network Δ.

Proposition 3.1 Suppose a nonmonotonic inheritance network Δ then

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

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

where E_i and E_j are extensions of Δ.

Proof See Appendix B. □

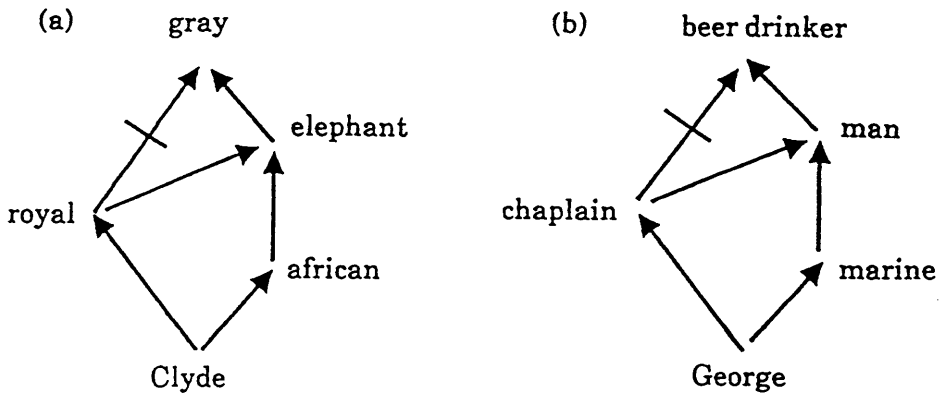
4 Discussion

The previous sections presented a formalization of the nonmonotonic inheritance network and its parallel execution algorithm. Our approach enables us to define inheritance rules apart from data in a network, and simplifies description and maintenance of the network. The problem is, as is mentioned in an earlier section, a redundant IS_A link often causes some ambiguities in the network.

The π algorithm produces a set of inheritable properties for an input class, and parallel execution is achieved in multiple inheritance. When a network is definite, the properties generated by the algorithm are included in an extension, while in the case of an indefinite network, it collects ambiguous information from multiple extensions. Note that the output of the algorithm is not an extension itself, thus there is no problem of logical inconsistency in the indefinite case. It may seem that such ambiguities give us no information, but they sugge. the problem in the network structure and help us to reconstruct it.

In general, however, it is not straightforward to decide whether an inheritance network is definite or not. Let us consider an example from [San86] (Figure (a)).

[San86] has defined some basic structure types for inheritance networks and given sound inference rules for these structures. According to his inheritance system, Clyde is not gray in the above network. Whereas Touretzky's system [Tou86] yields two alternatives; Clyde is gray in one case and is not in the other, and our system also returns gray and not(gray) for an input class Clyde. This shows that the above network is interpreted as definite in Sandewall's system, and indefinite in Touretzky's and ours. In this example, it seems to be more intuitive



to infer that Clyde is not gray. However, as in shown in [THT87], there is a counter-example which has the the same structure with (a), but the Sandewall's inference seems to be doubtful (Figure (b)). In this case, whether George drinks beer or not is not clear and it depends on the rate of beer drinking among marines.

These examples suggest that it is difficult to infer the intended meaning of the network from its structure alone.

Acknowledgment

We would like to thank David W. Etherington for his comments for improving earlier drafts of this paper. We are also grateful to the colleagues at ICOT for useful discussions and comments.

References

[Cot85] Cottrell,G.W.: "Parallelism in Inheritance Hierarchies with Exception", *IJCAI'85*, pp.194-202, 1985.

[ER83] Etherington,D.W. and Reiter,R.: "On Inheritance Hierarchies with Exceptions", *AAAI'83*, pp.104-108, 1983.

[Eth87a] Etherington,D.W.: "Formalizing Nonmonotonic Reasoning Systems", *Artificial Intelligence 31*, pp.41-85, 1987.

[Eth87b] Etherington,D.W.: "More on Inheritance Hierarchies with Exceptions", *AAAI'87*, pp.352-357, 1987.

[Fah79] Fahlman,S.E.: "NETL: A System for Representing and Using Real-World Knowledge", *MIT Press*, Cambridge, MA, 1979.

[HTT87] Horty,J.F., Thomason,R.H. and Touretzky,D.S.: "A Skeptical Theory of Inheritance", *AAAI'87*, pp.358-363, 1987.

[Rei80] Reiter,R.: "A Logic for Default Reasoning", *Artificial Intelligence 13*, pp.81-132, 1980.

[San86] Sandewall,E.: "Nonmonotonic Inference Rules for Multiple Inheritance with Exceptions", *Proc. of IEEE*, vol.74, pp.1345-1353, 1986.

[Tou86] Touretzky,D.S.: "The Mathematics of Inheritance Systems", *Research Notes in Artificial Intelligence*, Pitman, London, 1986.

[THT87] Touretzky,D.S., Horty,J.F. and Thomason,R.H.: "A Clash of Intuitions", *IJCAI'87*, pp.476-482, 1987.

[Ued86] Ueda,K.: "Guarded Horn Clauses", *Lecture Notes in Computer Sciences 221*, Springer-Verlag, Berlin, 1986.

Appendix A

Here, we show an implementation of the π algorithm in GHC. GHC is the parallel logic programming language developed as the kernel language of fifth generation project 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 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 the procedure is shown with an example of shellbearers.

```
/*** 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.

Appendix B

First, some notation used in the proof is given. For an $IS\_A(x, y)$, the closure of x is defined as a set, $Upper_k(x)$ $(k \geq 0)$, as follows:

1. $x \in Upper_0(x)$
2. $z \in Upper_{k+1}(x)$ iff $\exists y, y \in Upper_k(x)$ and $IS\_A(y, z)$.

Note that the IS_A hierarchy is acyclic, then $x \notin \bigcup_{k\geq 1} Upper_k(x)$. Now we show the proo of the proposition.

Proposition 3.1 Suppose a nonmonotonic inheritance network Δ then

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

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

where E_i and E_j are the extensions of Δ.

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 $\pi(c_n, Props_n)$ and $property(c_n, CProps_n)$ 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 $\forall c_k \in Upper_k(class)$ $(0 < k \leq n)$, $\pi(c_k, Props_k)$ where

$$Props_k = \{prop \mid \exists E_i, Property(c_k, prop) \in E_i\}$$
$$\cup\{not(prop) \mid \exists E_j, \neg Property(c_k, prop) \in E_j\} \text{ holds.}$$

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

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

where $property(c_{k-1}, CProps_{k-1})$.

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

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

(b) Otherwise, $uprop \in Temp_{k-1} - RevCProps_{k-1}$

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

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

By the assumption, $\forall c_k \in Upper_k(class)$, then

$\exists E_i, Property(c_k, uprop) \in E_i$ or $\exists E_j, \neg Property(c_k, uprop) \in E_j$.

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

$\neg Property(c_{k-1}, uprop) \notin W$ and $\exists E_i, Property(c_{k-1}, uprop) \in E_i$.

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

$Property(c_{k-1}, uprop) \notin W$ and $\exists E_j, \neg Property(c_{k-1}, uprop) \in E_j$.

Therefore,

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

While, let $\forall c_{k-1} \in Upper_{k-1}(class)$, $\exists E_i$, $Property(c_{k-1}, uprop) \in E_i$,

or $\exists E_j$, $\neg Property(c_{k-1}, uprop) \in E_j$.

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

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

(b) Otherwise, $\exists E_i, Property(c_k, uprop) \in E_i$, or $\exists E_j, \neg Property(c_k, uprop) \in E_j$.

By the assumption, $\forall c_k \in Upper_k(class)$, $uprop \in Props_k$,

or $not(uprop) \in Props_k$, where $\pi(c_k, Props_k)$.

In case $\exists E_i, Property(c_{k-1}, uprop) \in E_i, \neg Property(c_{k-1}, uprop) \notin W$ holds, then $uprop \notin RevCProps_{k-1}$ and $uprop \in Props_k - RevCProps_{k-1}$.

So, $uprop \in CProps_{k-1} \cup (Temp_{k-1} - RevCProps_{k-1})$.

In case $\exists E_j, \neg Property(c_{k-1}, uprop) \in E_j, Property(c_{k-1}, uprop) \notin W$ holds, then $not(uprop) \notin RevCProps_{k-1}$ and $not(uprop) \in Props_k - RevCProps_{k-1}$.

So, $not(uprop) \in CProps_{k-1} \cup (Temp_{k-1} - RevCProps_{k-1})$.

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

Therefore,

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

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

$$Props_{k-1} = \{prop \mid \exists E_i, Property(c_{k-1}, prop) \in E_i\} \cup \{not(prop) \mid \exists E_j, \neg Property(c_{k-1}, prop) \in E_j\}.$$

By induction, we have the desired result. □