The University of Alberta

Parallelism in Nonmonotonic Multiple Inheritance Systems

by

©

Vickitt Lau

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1988

ISBN 0-315-45619-1

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Vickitt Lau

TITLE OF THESIS: Parallelism in Nonmonotonic Multiple Inheritance Systems

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

(Signed) Lau Vickitt
Permanent Address:
Box 727, Sub 11
Edmonton, Alberta
Canada T5W 4A3

Dated 11 August 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Parallelism in Nonmonotonic Multiple Inheritance Systems** submitted by **Vickitt Lau** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]

Supervisor

[illegible]

[illegible]

[illegible]

Date August 11th, 1988

To my Parents and Aunt

Abstract

Organizing information in the form of an inheritance hierarchy is a basic concept in knowledge representation. Nonmonotonic multiple inheritance systems are the most powerful in their expressiveness. They allow a class to inherit properties from multiple superclasses and exceptions to those inherited properties are also permitted. However, due to their complexity, parallel processing using the Parallel Marker-Propagation Machine (PMPM) is difficult. In this thesis, we suggest two enhancements to the PMPM in order to make it more suitable for inheritance reasoning. In the first one, marker-value pairs are passed around instead of just markers. Such an enhancement makes it possible to have a completely parallel inheritance algorithm for unambiguous networks. In the second enhancement, a set of link-checking and link-setting commands are provided. By depositing relevant information to appropriate links, a serial-parallel algorithm can be designed for general inheritance networks.

Acknowledgements

I would like to thank my supervisor Dr. Jia You for his support of this research. I appreciate those numerous hours of insightful discussion with Dr. You when problems were discovered and solved. I would also like to thank members of my examining committe Dr. Reene Elio, Dr. Duane Szafron and Dr. Emil Girczyc for their valuable suggestions that make this a better thesis.

I am also grateful for the financial support from the Department of Computing Science and the Natural Science and Engineering Research Council of Canada.

Finally, thanks to Chris Heimers and Brian Wong for proof-reading my thesis and the moral support that Christina Lau has given me.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Introduction

One major goal of Artificial Intelligence (AI) is to build intelligent systems. In the early days of AI, it was believed that building intelligent systems hinged on a few simple and yet powerful techniques. Research in general search techniques and theorem proving flourished. However, after a decade or two of experimentation, it becomes clear that an intelligent system with only simple problem solving techniques is inadequate. An intelligent system has to have a large amount of knowledge which it can use to solve problems efficiently. This knowledge-based approach is the current paradigm of AI.

As AI systems try to solve more complex real-world problems, more real-world knowledge is required. The problem of how to handle a large amount of knowledge has been a persistent problem. There are two facets of the problem: how to represent the large quantities of real-world knowledge and how to search through the knowledge base quickly and effectively. As for knowledge representation, structuring knowledge in a form of hierarchy has emerged as a basic concept. As for effective retrieval of knowledge, parallelism is seen as a key to solving the problem.

1.2. Inheritance Systems

The basic idea of the hierarchical approach of knowledge representation is the taxonomy of structured description of knowledge. Concepts and entities of a knowledge base are organized in the form of an inheritance hierarchy. We can think of the concepts and entities as the nodes in the inheritance hierarchy. The relationship between the concepts and entities is represented by the links between the nodes. Various links are present in an inheritance hierarchy such as IS-A or positive link ($\rightarrow$) and IS-NOT-A or negative link ($\not\rightarrow$). If there is a positive link from node A to node B, then A is said to be a subclass of B. On the other hand, B is said to be a superclass of A and node A inherits all the properties of node B. Since the properties of the superclass do not need to be repeated, economy of information storage is achieved.

Familiar knowledge representation schemes such as semantic net and frame system incorporate properties inheritance. For instance, the semantic net shown in Figure 1.1 represents a knowledge base about vehicle. In Figure 1.1, the purpose of a vehicle is for transportation. Since we have an IS-A chain from sports car to vehicle, sports car inherits the purpose from vehicle, namely for transportation. The properties of a superclass are not explicitly represented in the subclasses. The purpose and the usage of a sports car are implicitly represented in the network and pieces of information are derived when needed. We call a representation system which incorporates such a concept an inheritance system. Well-known inheritance systems include FRL [Bobrow, Winograd 79], NETL [Fahlman 79] and KL-ONE [Brachman, Schmolze 85], to name just a few.

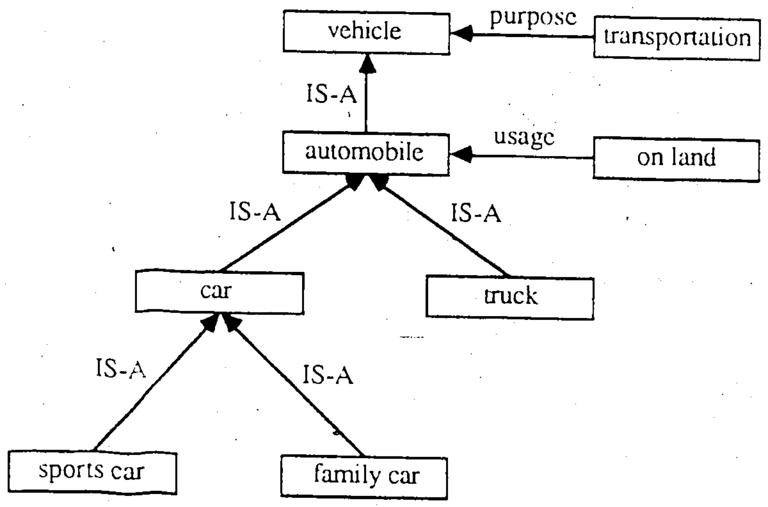


Figure 1.1 A semantic network: illustrating inheritance of properties

1.2.1. A Taxonomy of Inheritance Systems

Inheritance systems can be classified into different categories according to the types of links and structures that the system allows [Touretzky et al 87]. A **unipolar** system is one that allows only positive links. A **bipolar** system is one that allows both positive and negative links. It is obvious that negative statements can be expressed in a bipolar system but not in a unipolar system.

In a **tree-structured** inheritance system, the nodes can have at most one immediate ancestor. That is, each node can inherit properties from only one immediate ancestor. In contrast, the nodes in a **multiple** inheritance system can have more than one immediate ancestor.

We can also distinguish between a **nonmonotonic** inheritance system and a **monotonic** inheritance system. In a nonmonotonic inheritance system, exceptions to inherited properties are allowed; while in a monotonic inheritance system, a descendant has to inherit all the properties of its ancestors.

Finally we say an inheritance system is **homogeneous** if it is either monotonic or thoroughly nonmonotonic. By thoroughly nonmonotonic, we mean that every link in the system is defeasible, i.e., can be overriden by another link. In contrast, a strict link is one which cannot be overriden. In a **heterogeneous** inheritance system, both strict and defeasible links are present. [Brachman 85] discusses the need for such systems.

Reasoning with a monotonic inheritance system is quite straightforward. However, this kind of systems is not very interesting because a lot of the real-world knowledge that we want to encode in an inheritance system is normative, that is, statements that allow for exceptions. Reasoning with a tree-structured inheritance system is also quite straightforward. But the application of such systems is limited due to the fact that each node can only inherit properties from one immediate ancestor.

Nonmonotonic multiple inheritance systems are much more general but at the same time much more difficult to deal with. Firstly, the semantics of such systems is not very clear. Secondly, when the semantics is clarified, how to do parallel inference in such systems becomes a problem because of their complexity. This thesis studies the problems concerning nonmonotonic multiple inheritance systems; in particular, how to make parallel inference in such inheritance systems. Throughout the thesis, an inheritance network or hierarchy is used as a short-hand for nonmonotonic multiple inheritance network.

1.2.2. Common-sense and Inheritance Reasoning

The nonmonotonic nature of common-sense reasoning has prompted recent interest in developing systems for nonmonotonic reasoning. Traditional reasoning systems are monotonic in the sense that the addition of new information will not invalidate old information. The familiar example about the flying ability of Fred demonstrates the point: if we know that Fred is a bird, we would assume that it can fly. However, if we discover that Fred is a penguin upon further investigation, we would conclude that Fred cannot fly. The old information about Fred being a flying bird is cancelled when we have the additional information that it is a particular kind of bird, namely, a penguin. The problem here is that we are always forced to make inference based on only partial information. But when more complete or more specific information arrives, we may want to draw different conclusions.

Common-sense reasoning, in general, is difficult. Inference in an inheritance hierarchy with exceptions can be viewed as an instance of common-sense reasoning. The flying bird example above, for instance, can be viewed as inheritance reasoning. Some other forms of common-sense reasoning, however, do not fall into the category of inheritance reasoning. For instance, the following rule is called default assignment [Reiter 78]: "unless you know otherwise, assume that a person's hometown is that of his/her spouse." Another form of common-sense reasoning which also involves nonmonoticity is the closed world assumption [Reiter 78], such as "unless you know otherwise, assume that Mary is not a student." These forms of common-sense reasoning are rather common. Nonetheless, they do not exhibit the hierarchical nature of inheritance reasoning. Formalisms, such as Default Logic [Reiter 80], which attempt to capture common-sense reasoning have serious computational problems. Inference in Default Logic, for instance, is not even semi-decidable in general. On the other hand, due to its hierarchical nature, inheritance reasoning is easier to handle compared with general common-sense reasoning and more efficient reasoning algorithms can be designed. In addition, a large amount of real-world knowledge does exhibit a hierarchical nature. All these make the inheritance system an interesting and worthwhile topic for study.

1.2.3. The Inheritance Principle

The most important notion of an inheritance system is that subclass should override superclass, i.e. the information provided by a subclass has higher priority than that of a superclass. For example, if A inherits property $\sim P$ from B and property P from C, then A should have property $\sim P$ if B is a subclass of C (see Figure 1.2). We call this the inheritance principle.

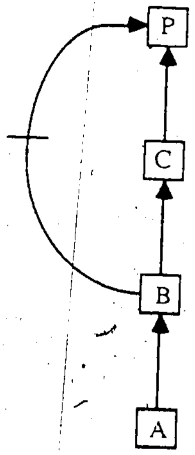


Figure 1.2 Network illustrating the inheritance principle

Suppose we substitute A for *fred*, B for *penguin*, C for *bird* and P for *fly* in Figure 1.2, then it is clear why the inheritance principle should be followed. The reason is that *penguin* is a subclass of *bird*, information provided by *penguin* is more specific than that provided by *bird*. In making our decision whether *fred* can fly or not, we prefer the more specific information. The idea that information provided by a subclass should override those of a superclass is adapted by all inheritance systems such as FRL [Bobrow, Winograd 79] and NETL [Fahlman 79]. However, this intuitive notion is not formalized until recently by [Touretzky 86]. We will discuss the need to formalize this important intuitive notion in Section 1.4.

1.3. Parallel Architecture for Inheritance Systems

For an AI system to deal with any sufficiently complex real-world problems, we can expect it to have a rather large knowledge base. However, we do not want the size of the knowledge base to hinder the performance of the overall system. This leads to interests in designing parallel architecture for inheritance systems.

[Fahlman 79] describes a representation system called NETL for inheritance systems and the Parallel Marker Propagation Machine (PMPM) for parallel inferencing in an inheritance network. The essential idea is to make each node in the inheritance network correspond to some hardware processing element and the relationship between the two nodes in the inheritance network correspond to a hardware link between the hardware elements. The hardware processing elements and the hardware links are conveniently called nodes and links respectively. The PMPM is a Single Instruction Stream, Multiple Data Stream (SIMD) machine [Hwang, Briggs 85]. Both the processing elements and the links respond to commands issued by a central controller. The purpose of the commands is to propagate marker bits between the nodes. The result of a sequence of marker propagation commands lies in the markers that the nodes receive. The primary types of inference supported by such a machine are property inheritance, transitive closure, and set intersection.

Inheritance reasoning in a tree-structured monotonic inheritance system is equivalent to finding the transitive closure of the IS-A link. This can be done by the PMPM in time proportional to the depth of the given network. However, inheritance reasoning in a multiple nonmonotonic inheritance system is much more involved and parallel algorithms are more difficult to design.

1.4. Formalizing Inheritance Systems

1.4.1. Meaning of IS-A Link

The meaning of an IS-A link depends on the kind of inheritance systems we are talking about. In a monotonic multiple inheritance system, each link can be "logically" described [Hayes 79]. A link such as $A \rightarrow B$ where A and B are classes can be described by the first-order logical formula $A(x) \Rightarrow B(x)$. A link such as $a \rightarrow A$ where a is an individual and A is a class, can be described as $A(a)$. This simple translation between the links in a network and first-order formulae is possible only because the links are not defeasible, i.e. cannot be cancelled. When exceptions are allowed, the formula $A(x) \Rightarrow B(x)$ does not accurately describe what we want to say about $A \rightarrow B$ any more.

An IS-A link in a nonmonotonic multiple inheritance system has a totally different meaning than that in a monotonic multiple inheritance system. Suppose we have $bird \rightarrow fly$ in a network. Intuitively, this means "bird flies". We can think of a number of equally likely interpretations of this kind of generic sentence [Carlson 82]:

(1) All birds fly

(2) Typical birds can fly

(3) Most birds can fly

(4) If we find a bird, it is likely that it can fly

(5) If we find a bird, assume it can fly unless we know that it cannot

Strictly speaking, interpretation (1) is incorrect since we allow exceptions to the general rule. (2) reflects more accurately what we want to say about birds. But the problem is that we need a definition of typicality. Interpretation (3) needs a vague quantifier "most". In different generic statements, the quantifier has different strength. [Rich 83] favors interpretation (4). She argues that common-sense or default reasoning is actually a form of likelihood reasoning. But the problem is that we need to specify how likely it is that a bird we find will be able to fly. Specifying that likelihood seems to be arbitrary. (5) sounds more nonmonotonic than others. "birds fly" is interpreted as a default rule which says given a bird, we assume it can fly by default but if more specific information arrives, we are willing to retract the previous assumption.

1.4.2. The Shortest Path Algorithm

Early inheritance systems, such as NETL, suffered from lack of precise semantics. No matter how fast we can make inference within an inheritance system and how expressive an inheritance system is, a lack of precise semantics means that we do not understand what we are doing with the inheritance system and we do not even know what we are talking about when we try to assert some knowledge using the inheritance system. Formal semantics is clearly desirable.

The behavior of those early inheritance reasoning algorithms is shown by giving a few examples. These examples are usually relatively simple and always show the "desired" behavior of the system. The trouble is that for real-world problems, which are far more complex, the algorithm used for inferring inheritance may not behave as expected. Sometimes it may even produce wrong results. In other words, we cannot claim that we understand an inheritance algorithm or that an algorithm is correct only by tracing the behavior of an algorithm using a few examples.

Part of the reason that no rigorous semantics is given for multiple inheritance systems is that the inheritance principle seems to be so intuitive: subclasses should override superclasses. This principle is translated into something like the following. Suppose that we want to determine whether node A is a subclass of node B. Suppose also that there are two paths from A to B; one allows us to conclude that A is a subclass of B while the other allows us to conclude that A is not a subclass of B. Which one should we choose? Intuitively, we would want to choose the shorter path. For instance, suppose we know that birds are flying things, penguin is a bird, fred is a penguin and that penguin cannot fly. Graphically, it is represented as in Figure 1.3:

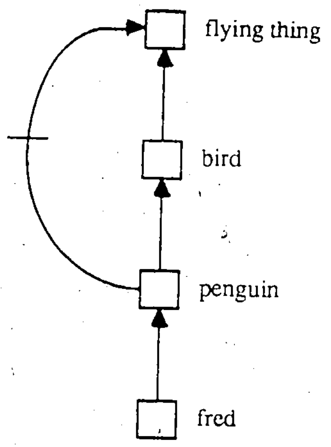


Figure 1.3 Network that the shortest path algorithm works

The path *fred* $\rightarrow$ *penguin* $\rightarrow$ *bird* $\rightarrow$ *flying thing* is of length three while the path *fred* $\rightarrow$ *penguin* $\not\rightarrow$ *flying thing* is of length two. The shorter path, in this case, overrides the longer path and we have the desired conclusion: fred cannot fly. Unfortunately, this shortest path heuristic is not always correct. The example in Figure 1.4 is shown in [Touretzky 86]:

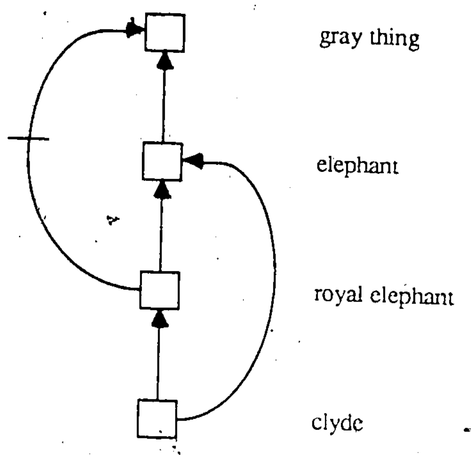


Figure 1.4 Network with a redundant link

In this example, a redundant link *clyde* $\rightarrow$ *elephant* is present. We say that *clyde* $\rightarrow$ *elephant* is redundant because we can conclude that *clyde* is an *elephant* without the explicit link. The two paths *clyde* $\rightarrow$ *elephant* $\rightarrow$ *gray thing* and *clyde* $\rightarrow$ *royal elephant* $\not\rightarrow$ *gray thing* are both of length two. Although it seems to be an ambiguous situation, intuitively we want to conclude that *clyde* is not gray. The reason is that without the redundant link, we would conclude the same. Another problem with this shortest path heuristic is shown in Figure 1.5.

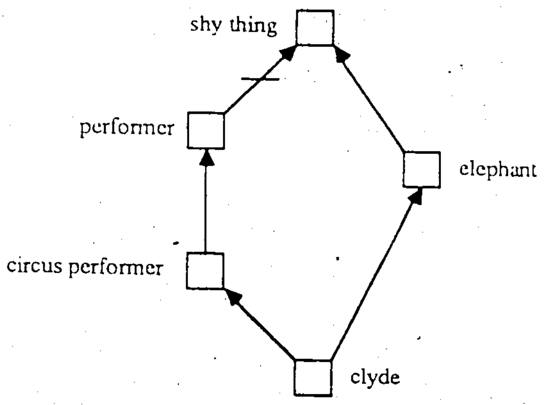


Figure 1.5 An ambiguous network

Although the path *clyde* $\rightarrow$ *elephant* $\rightarrow$ *shy thing* is shorter than *clyde* $\rightarrow$ *circus performer* $\rightarrow$ *per-*

former $\not\rightarrow$ *shy thing*, we have no reason to allow the first path to override the second path. Such a network is called an ambiguous network [Touretsky 86]. Thus when redundant links are present or the network is ambiguous, the shortest path algorithm is not very useful.

As we can see, although the notion of the inheritance principle is intuitive, it is not as simple as the inheritance system designers once thought. The shortest path algorithm seems to capture the intuitive notions naturally. Upon careful consideration, however, we see that it is not adequate to accomplish the task. Formalizing this intuitive notion allows us to say precisely wh we mean by subclass overriding superclass.

1.5. Scope and Overview of the Thesis

This thesis investigates the problems related to parallel architectures and algorithms for nonmonotonic multiple inheritance systems. Recently, there has been an explosion of attempts to formalize inheritance reasoning. Formalizing inheritance reasoning allows us to say precisely what inheritance is all about. But different formalisms may have different opinions as to what inheritance should mean. In Chapter 2, we survey the different formalisms for inheritance reasoning. Since there is a host of different formalisms for inheritance reasoning, it is important to stick with one particular formalism when we design parallel architecture and algorithm. The goal of this thesis is to design parallel architecture and algorithm for inheritance reasoning as defined in [Touretzky 86].

There are a number of open problems related to parallel architectures and algorithms for inheritance reasoning [Etherington 87a]:

(1) Are there natural classes of inheritance networks with exceptions which admit parallel inference algorithms yet do not preclude the representation of our common-sense knowledge about taxonomies.

(2) Define parallel architectures and algorithms for inheritance reasoning and prove their

correctness.

The PMPM described in Section 1.3 is too simple as a parallel architecture for inheritance reasoning for nonmonotonic multiple inheritance networks. Although it supports the shortest path algorithm for property inheritance naturally, we have seen in the previous section that the algorithm is not reliable when the given network has redundant links or when ambiguity is involved. Despite its limitation, we do not need to discard the PMPM as a parallel architecture for inheritance reasoning. There are several attempts to circumvent the inadequacy of the PMPM. In Chapter 3, we give a survey of these attempts.

All the attempts described in Chapter 3 use the PMPM and then try to find ways to do parallel or quasi-parallel processing in an inheritance network. In this thesis, we take another approach. Since the problems of using the PMPM to deal ~~with~~ inheritance reasoning stems from its oversimplicity, it is natural to consider ways to enhance its power so that it is more suitable as a parallel architecture for inheritance reasoning. Two different enhancements of the PMPM are introduced in this thesis. In the first one, each node is allowed to pass around an additional number, called the level number. Using this additional information, we design a one-pass parallel inheritance algorithm for unambiguous networks. In the second enhancement, links can be marked by special link markers. By depositing relevant information to such links, we design a serial-parallel inheritance algorithm capable of dealing with general inheritance networks. Chapter 4 and 5 discuss these enhancements and algorithms respectively. Finally in Chapter 6, we summarize the thesis and discuss some future research.

Chapter 2

Formalizing Inheritance Reasoning

2.1. Introduction

In this chapter, we survey a number of different attempts to formalize inheritance reasoning. Formal systems such as Default Logic [Reiter 80], Nonmonotonic Logic [McDermott and Doyle 80] and Circumscription [McCarthy 80] attempt to formalize common-sense reasoning. Since there is a close relationship between inheritance and common-sense reasoning, it is not surprising that these systems are also used to formalize inheritance reasoning. The problem is that the inheritance principle, which is the most important notion of inheritance reasoning, is not captured in these formalisms. [Poole 85] proposes a theory comparator to capture the inheritance principle inside the semantics of the system. Although common-sense and inheritance reasoning are closely related, the hierarchical nature of inheritance reasoning separates them. This makes inheritance systems an interesting subject for study independent of common-sense reasoning. [Touretzky 86] uses a mathematical approach to give a precise formalization of the inheritance principle. Finally [Sandewall 86] attempts to incorporate the inheritance principle into the inference rules of a nonmonotonic system.

2.2. Default Logic

The first attempt to formalize nonmonotonic multiple inheritance systems is done by [Etherington and Reiter 83]. Their approach is to establish a correspondence between nonmonotonic inheritance hierarchies and Default Logic [Reiter 80]. Links in a hierarchy are identified as either default rules or first-order formulae. Furthermore, exceptions to default rules are explicitly specified by using another exception link. In such a way, a formal semantics for nonmonotonic inheritance hierarchies is provided based on Default Logic.

A default theory Δ consists of an ordered pair (D, W) where D is a set of defaults and W is a set of first-order formulae. A default is an expression of the following form:

$$\frac{A(x) : B_1(x), \ldots, B_n(x)}{C(x)}$$

where $A(x)$, $B_1(x)$, . . . , $B_n(x)$, $C(x)$ are first-order formulae. The interpretation of the above expression is: if you believe that $A(x)$ is true and it is consistent to assume $B_1(x)$ through $B_n(x)$, then you are entitled to conclude that $C(x)$ is true. A is called the prerequisite of the default, B_i's are the justifications and C is the consequent of the default.

A default theory may have zero, one or more extensions. An extension can be thought of as a set of first-order formulae which is the result of extending or expanding W by the set of defaults D of Δ. In this sense, defaults are used to provide a more complete database on the incomplete database W. An extension can be formally defined. Before doing that, a few preliminary definitions are needed. Let L denote the set of first-order well-formed formulae (*wff*). A *wff* containing no free variable is said to be *closed*. In the default rule shown above, if $A(x)$, $B_1(x)$, . . . , $B_n(x)$ and $C(x)$ contain no free variable, the default is said to be closed. If each default in D is closed, then Δ is called a closed default theory. The logical closure $TH_L(S)$, given an arbitrary set of *wffs* S, is defined as:

$$TH_L(S) = \{ w \mid w \in L, w \text{ is closed}, S \vdash w \}$$

An extension for a closed default theory $\Delta = (D, W)$ is defined as a minimal fixed point of an operator Op having the following properties:

(1) $W \subseteq Op(S)$

(2) $TH_L(Op(S)) = Op(S)$

(3) For each default, $\dfrac{A(x) : B_1(x), \ldots, B_n(x)}{C(x)}$, if $A(x) \in Op(S)$ and $\sim B_1(x), \ldots, \sim B_n(x) \notin Op(S)$, then $C(x)$ is in $Op(S)$

Consider the following default t[illegible] $\Delta = (D, W)$ where

$$D = \{ \frac{A(x):B(x)}{B(x)}, \frac{B(x):A(x),C(x)}{C(x)}, \frac{C(x):\sim A(x),B(x)}{D(x)} \} \quad \text{and}$$

$$W = \{A(a)\}.$$

Using $A(a)$ and the first default rule in D, both $A(a)$ and $B(a)$ are in the extension. Applying $A(a)$, $B(a)$ in the second default rule in D, the extension is incremented to $A(a)$, $B(a)$, and $C(a)$. The third default rule cannot be applied since $A(a)$ is in the extension. No more elements can be added to the extension and therefore the final extension $E = \{A(a), B(a), C(a)\}$.

The above default theory has a unique extension. Not all default theories have unique extensions. For example, a default theory $\Delta = (D, W)$ where

$$D = \{ \frac{A(x):B(x)}{\sim C(x)}, \frac{A(x):C(x)}{\sim B(x)} \} \quad \text{and}$$

$$W = \{A(a)\}$$

has two extensions: $E_0 = \{A(a), \sim C(x)\}$ and $E_1 = \{A(a), \sim B(x)\}$. These extensions can be considered as consistent sets of belief that are derived from the default theory Δ.

Speci[illegible] cases of defaults exist which have some interesting properties. For example, defaults of the form:

$$\frac{A(x):B(x)}{B(x)}$$

are called normal defaults. A default theory Δ in which D consists of normal defaults only is guaranteed to have at least one extension. This is a desirable property of a default theory because if it does not have even one extension, that means we cannot generate a consistent set of beliefs from it and the default theory is quite useless. Although normal defaults have this nice property, they are not sufficient for formalizing inheritance [Reiter, Crisuolo 81]. Consider the example shown in Figure 2.1. If the links in the network are represented by normal defaults, we have:

(1) $\dfrac{bird(x) : fly(x)}{fly(x)}$

(2) $\dfrac{penguin(x) : bird(x)}{bird(x)}$

(3) $\dfrac{penguin(x) : \sim fly(x)}{\sim fly(x)}$

Finally, the starting node of the network is represented by the first-ordered formula *penguin(fred)*. From above, two equally valid conclusions can be generated: *fly(fred)* and *~fly(fred)*. The first conclusion is drawn by using defaults (1) and (2) together with the fact that *fred* is a *penguin*. The second conclusion is derived from default (3) and the final fact. In an inheritance hierarchy, however, we want to conclude that fred cannot fly since the other conclusion is derived from a more general rule.

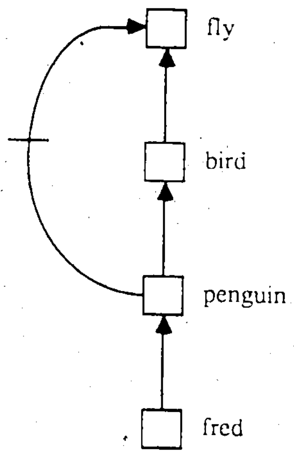


Figure 2.1 Network showing fred cannot fly

Notice that penguin is actually an exception to the general rule that birds can fly. To handle this competition among defaults, it is suggested that exceptions be made explicit. Instead of the above set of defaults, the network in Figure 2.1 may be represented by the following default theory:

(1) $\frac{bird(x) : fly(x) \wedge \neg penguin(x)}{fly(x)}$

(2) $\frac{penguin(x) : bird(x)}{bird(x)}$

(3) $\frac{penguin(x) : \neg fly(x)}{\neg fly(x)}$

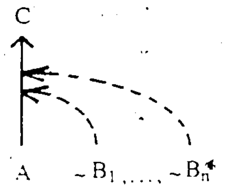
(4) $penguin(fred)$

The default th[illegible]ry will generate only the desired conclusion $\neg fly(fred)$. This is because the application of default rule (1) is suppressed due to the fact that *fred* is a *penguin*. This form of default, which allows exceptions to be made explicit, is called semi-normal default:

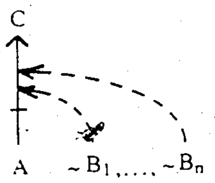
$$\frac{A(x) : B_1(x), \ldots, B_n(x), C(x)}{C(x)}$$

The characteristic of semi-normal default is that the justification of the default rule entails the consequent but not vice versa. Unlike normal default theory, semi-normal default theory is not guaranteed to have any extension. [Etherington 87b], however, shows that a subclass of semi-normal default theory, called ordered semi-normal default theory, does have at le[illegible]e extension. Ordered semi-normal default theories actually correspond to acyclic networks. Graphically, semi-normal defaults are represented as follows:

$$\frac{A(x) : \neg B_1(x), \ldots, \neg B_n(x), C(x)}{C(x)}$$



$$\frac{A(x) : \neg B_1(x), \ldots, \neg B_n(x), \neg C(x)}{\neg C(x)}$$



In this formalization, strict links and default links are also distinguished. In a network, they are represented as:

$$\frac{A(x) : B(x)}{B(x)}$$

$$\begin{tikzcd} B \\ A \arrow[u] \end{tikzcd}$$

$$A(x) \Rightarrow B(x)$$

$$\begin{tikzcd} B \\ {} \arrow[u, -latex] \end{tikzcd}$$

So the network shown in Figure 2.1 is represented by this formalism as the network in Figure 2.2.

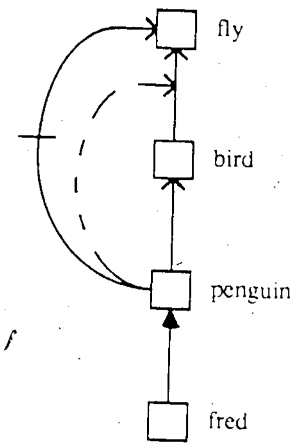


Figure 2.2 Network represented using the Default Logic Formalism

As mentioned before, there has been no attempt to give a precise formalization for inheritance hierarchies. As a result, there is no guideline to determine whether the inference produced by an inference algorithm is correct or not. The notion of correctness is formalized in [Etherington, Reiter 83] as the ability of an inference algorithm to derive conclusions that lie within a single extension of a given default theory.

An inference algorithm is also presented which is shown to be correct according to the above definition of correctness. The algorithm is actually capable of constructing all the extensions of an arbitrary finite default theory. A default theory is said to be finite if it has only finitely many variables, constant and predicate symbols, and defaults [Etherington 82]. The algorithm is based on a so-called "relaxation style constraint propagation technique". An extension is constructed by a number of steps. At each step, we generate an approximation of an extension by applying an unused and applicable default. A default is applicable at step $i+1$ if its prerequisites are "known" and its justifications are consistent with the approximations at steps i and $i+1$. The algorithm proceeds to the next approximation when there is no more default that is applicable. When the approximation at step $i+1$ is the same as that of step i, the algorithm is said to converge, and the approximation is an extension of the default theory.

[Etherington 87a] noted that there are some finite default theories which admit nonconverging computations. However, a subclass of finite default theory is defined such that the algorithm, when applied to such a theory, is more well-behaved. It is called an ordered network theory. It is stated as a theorem that for finite ordered theories, the procedure given (in [Etherington 87a]) always converges on an extension.

2.2.1. Discussion

As has been mentioned, a basic notion of inheritance hierarchies with exceptions is that subclasses should override superclasses. From the above description, it is clear that this intuitive notion is not captured by representing inheritance in Default Logic. [Touretzky 84] complains that since this intuitive notion is not captured, this formalization does not fully express the meaning of inheritance.

Another characteristic of this Default Logic formalism is that exceptions have to be made explicit in order to control interactions among defaults. [Touretzky 84] observes that this requisite is rather demanding in the sense that modifications of defaults already present in the knowledge base have to be made as we add new information. This modification is a complex task when the knowledge base is large. In addition, individual defaults also become increasingly complex as the knowledge base grows. Finally, the translation of an individual link in the Default Logic formalism is not independent of other links in the network. In other words, representing inheritance by semi-normal defaults lacks representational conciseness and does not facilitate simple modification of the existing network.

2.3. Nonmonotonic Logic

Nonmonotonic Logic developed in [McDermott, Doyle 80] uses a sentential operator M, instead of an inference rule, to deal with default reasoning. The modal operator M used in the system is taken to mean "is logically consistent". The network in Figure 2.1 can be expressed as sentences in nonmonotonic logic:

(1) $bird(x) \wedge M[fly(x)] \Rightarrow fly(x)$

(2) $penguin(x) \wedge M[bird(x)] \Rightarrow bird(x)$

(3) $penguin(x) \wedge M[\sim fly(x)] \Rightarrow \sim fly(x)$

(4) $penguin(fred)$

As we can see, these sentences are very similar to the normal defaults in Default Logic, and we can expect the same problem with nonmonotonic logic as in Default Logic, namely that exceptions have to made explicit. It is more desirable for a formalism to capture the inheritance principle implicitly rather than explicitly.

2.4. Theory Preference

In the Default Logic formalism we associate multiple extensions to an inheritance hierarchy. As long as an inheritance algorithm produces conclusions that lie within a single extension, we say that it is correct. Nevertheless, in inheritance reasoning, we do not treat all extensions equally. Due to the inheritance principle, we prefer some extensions to others. For example in Figure 2.3, both $E_0 = \{A, B, C\}$ and $E_1 = \{A, B, \sim C\}$ are extensions. But the inheritance principle tells us that we want to choose E_1 instead of E_0. The Default Logic formalism says nothing about this theory preference. [Poole 85] discusses a system in which a theory comparator can be used to compare different theories that explain a sentence. With this theory comparator, the more specific theory can be chosen as the preferred theory and hence the inheritance principle is captured implicitly.

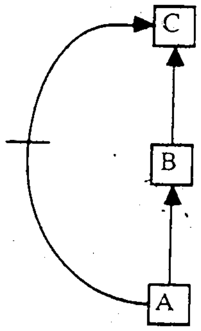


Figure 2.3 Extension $\{A, B, \sim C\}$ is preferred

In terms of Default Logic, the system described by Poole uses only normal defaults. The form of defaults is <*list_of_variables*> *Assume* <*wff*> where *wff* is a first-order well-formed formula. Besides defaults, facts can also be expressed in the form of a well-formed formula. An *instance* of a default is a default in which the variables in *list_of_variables* are substituted by actual values.

Let F be a set of well-formed formulae or facts and Δ be a set of defaults. Let D denote a set of instances of the elements of Δ. A well-formed formula g is said to be *explainable* if there is some D such that $F \cup D \models g$ and $F \cup D$ is consistent. D is then said to be a *theory* that *explains* g.

The set F is further divided into two sets: the set of necessary facts F_n and the set of contingent facts F_c. Graphically, F_n corresponds to the links in the network while F_c corresponds to the starting nodes of the network. The reason for dividing a set of facts into F_n and F_c is that we want to be able to reason with alternatives. For necessary facts, there are no other alternatives. But for contingent facts, we want to reason with the case when something else is true. Given F_n, F_c and Δ, a *solution* S is a pair $<D, g>$ where D is a theory that explains g. Formally we have $F_n \cup F_c \cup D \models g$. Let F_p be some possible fact. F_p is said to be *adequate* to make D explain g if $F_p \cup F_n \cup D \models g$. A solution $S = <D, g>$ is said to be *applicable* if $F_p \cup F_n \cup D \models g$.

The theory comparator is defined as follows. Given two solutions $S_1 = <D_1, g_1>$ and $S_2 = <D_2, g_2>$, S_1 is more specific than S_2 ($S_1 \geq S_2$), if for all F_p,

$$F_p \cup D_2 \cup F_n \not\models g_1 \text{ and } F_p \cup D_1 \cup F_n \models g_1 \text{ then } F_p \cup D_2 \cup F_n \models g_2$$

That is, if for all F_p which is not adequate to make D_2 explain g_1 but is adequate to make D_1 explain g, then F_p is also adequate to make D_2 explain g_2. Solution S_1 is said to be strictly more specific than S_2 ($S_1 > S_2$) if $S_1 \geq S_2$ and $S_2 \not\geq S_1$. In an inheritance hierarchy, we prefer the most specific solution.

Again, let us consider the network shown in Figure 2.1. The network can be represented as follows:

(1) (x) *Assume* $bird(x) \Rightarrow fly(x)$

(2) (x) *Assume* $penguin(x) \Rightarrow \neg fly(x)$

(3) (x) *Assume* $penguin(x) \Rightarrow bird(x)$

(4) $penguin(fred)$

With these, $fly(fred)$ can be explained by the theory $D_1 = \{bird(fred) \Rightarrow fly(fred)\}$ and $\neg fly(fred)$

by the theory $D_2 = \{penguin(fred) \Rightarrow \neg fly(fred)\}$. Theory D_2 is more specific than D_1 because there is an F_p, namely $bird(fred)$ which can make the solution $<D_1, fly(fred)>$ applicable but $<D_2, \neg fly(fred)>$ not applicable. Furthermore, D_2 is strictly more specific than D_1 because we cannot show that D_1 is more specific than D_2. We show this by considering the contingent facts which make $<D_2, \neg fly(fred)>$ applicable. There can only be two such facts: $penguin(fred)$ and $\neg fly(fred)$. Obviously, $\neg fly(fred)$ is adequate to make D_1 explain $\neg fly(fred)$. Therefore this contingent fact cannot be used to show that D_1 is more specific than D_2. For the other contingent fact, $<D_2, fly(fred)>$ is also applicable, which also means we cannot use this to show that D_1 is more specific that D_2. Since those two are the only contingent facts that can make $<D_2, \neg fly(fred)>$ applicable, and that they do not allow us to conclude that D_1 is more specific that D_2, D_1 is not more specific than D_2. Therefore D_2 is strictly more specific than D_1 and we prefer the conclusion $\neg fly(fred)$ which is drawn from theory D_2.

2.5. Circumscription

Another formalism that attempts to capture common-sense reasoning is Circumscription [McCarthy 80]. Circumscription was proposed as a rule of conjecture when we need to make a decision upon incomplete information. The argument for such a rule of conjecture is that in real life, even simple axioms need a very long list of qualifications. For instance, the rule "bird flies" cannot be applied in a general situation since there are a number of species of bird that cannot fly. Enumerating these species still would not work since we may encounter some unexpected birds such as wingless bird or one-winged bird etc. The idea of circumscription is to augment the set of axioms of the given domain with a "circumscription formula" so that the set of axioms and the formula together will limit the objects that satisfy a particular predicate to just those that the domain says must satisfy that predicate.

For example, suppose we have a sentence: $fly\_thing(A) \wedge fly\_thing(B)$. Circumscribing the predicate fly_thing in the above formula will result in

$$\forall x \; fly\_thing(x) \Leftrightarrow x{=}A \lor x{=}B$$

which says the only objects that are *fly_thing* are A and B. More general forms of circumscription also exist, such as letting some predicates vary in addition to those being minimized. This results in a circumscription formula which is even more restrictive than normal circumscription. The reason is that during such a process, we are willing to allow different objects to satisfy the variable predicate, in order to find the minimal number of objects which satisfy the predicate in question.

[McCarthy 86] describes how common-sense reasoning, in particular, default reasoning can be formalized by using circumscription. Recall the inconvenience of the Default Logic formalism in which exceptions have to be made explicit. The axioms in the database have to be changed when new information arrives. McCarthy suggests that an abnormal predicate, *ab*, can be used to more efficiently describe the abnormality of certain objects with respect to a particular aspect. For instance, we can describe the general rule that birds fly by the following formula:

$$bird(x) \land \neg ab(x) \Rightarrow flies(x)$$

The formula says that all birds can fly except those that are abnormal. In here, we only consider the case where there is only one kind of abnormality. In general, an object may be abnormal in a number of aspects. This can be represented by something like $abnormal(aspect1(x))$, $abnormal(aspect2(x))$, etc, which says x is abnormal with respect to aspect 1 and aspect 2. When exception to the above general rule is discovered, the general rule does not need to be changed. Instead, only one axiom needs to be added that says the exceptional object is abnormal, for instance:

$$penguin(x) \Rightarrow ab(x)$$

McCarthy calls the above a *cancellation of inheritance* axiom. One result of this formalism is that the different aspects of abnormalities are being made as part of the inheritance hierarchy.

The network of Figure 2.1 can be expressed by the following axioms:

(1) $penguin(fred)$

(2) $penguin(x) \Rightarrow bird(x)$

(3) $penguin(x) \Rightarrow ab(x)$

(4) $bird(x) \wedge \neg ab(x) \Rightarrow fly$

Circumscribing ab by letting fly to vary we get the circumscriptive formula:

$$ab(x) \Leftrightarrow penguin(x)$$

which allows us to conclude that penguin cannot fly.

Similar to the Default Logic formalism, formalizing inheritance reasoning by circumscription also does not capture the inheritance principle. Although it is more convenient to add new axioms in this system, exceptions still need to be explicitly mentioned. Another difficulty with this formalism is how to compute the circumscriptive formula.

2.6. Mathematical Formalism of Inheritance

The central concern in this mathematical approach [Touretzky 86] is to give a precise formalization of the intuitive notion of subclasses overriding superclasses in an inheritance network. Unlike the previous approaches in which the links of an inheritance system are analyzed as some assertions in a logical system, Touretzky chooses to analyze inheritance systems directly in terms of the links in the network. Inference in such a system can be thought of as constructing inference paths between the nodes.

Touretzky wants to find an ordering in which subclasses are allowed to override superclasses. An ordering which is based on the length of competing inference paths is obviously inadequate. Such an ordering corresponds to the shortest path algorithm which can be shown to derive erroneous results.

[Touretzky 86] proposes a partial ordering called the inferential distance ordering which has the desired properties. Traditional path length ordering orders the nodes according to the length of the path between two nodes. The inference distance ordering, on the other hand, orders the nodes according to the "between-ness" of two nodes. Suppose that A can inherit property P from node C and $\sim P$ from node B. That is, there are two inference paths p_1 and p_2 from A to P which allow us to draw conflicting conclusions. Using the inferential distance ordering, we let A inherit property P from C if there is an inference path from C to B and not vice versa. On the other hand, if there is an inference path from B to C and not vice versa, then we let A inherit property $\sim P$ from B. If none of the above is true, then we have an ambiguous situation. The reasoning behind this ordering is that if there is an inference path from B to C and not vice versa, then node B can be viewed as a subclass of C. Since subclass should override superclass, A should inherit property $\sim P$ from node B. Consider Figure 2.1. Since there is an inference path from *penguin* to *bird*, *penguin* is a subclass of *bird*. Therefore *penguin* provides more specific information and *fred* should inherit the property $\sim fly$ from it.

The path length ordering and the inferential distance ordering coincide when the network is tree-structured. But for multiple inheritance system, the two orderings differ when redundant links are present or when the given network is ambiguous.

Since the later study of this thesis is based on Touretzky's formalism, we now give more detailed definitions. A network is specified by a set of inheritance assertions or links. Since a link is formed by two nodes, we say that a link has a length of two. An inheritance path denoted by $<x_1, x_2, \ldots, x_n>$, is a sequence of nodes of length ≥ 2. Nodes are signed. They can be of the form $+x_i$, $-x_i$ and $\#x_i$ which are referred to as tokens. A link of the form $<+x,+y>$, $<+x,-y>$ and $<+x,\#y>$ says x is a y, x is not a y, and no conclusion whether x is y respectively. A path

$<x_1, x_2, \ldots, x_n>$, in which $x_1, \ldots, x_{n-1}$ are all positive is said to be well-formed. In the definitions below, the following notations are used:

Π: the set of all individuals and predicates in an inheritance system.

Θ: the set of all tokens derivable from Π, i.e., $\Theta = \{+,-,\#\} \times \Pi$.

Σ: the set of all sequences in Θ^* of length at least two.

Γ: the set of well-formed order pairs of tokens, i.e., $\Gamma \subseteq \Theta \times \Theta \subseteq \Sigma$.

σ: an element of Σ.

Φ, S: subsets of Σ.

x': tokens that match except for sign, such as, if the sign of x is +, then x' can be $-x$ or $\#x$.

The *conclusion set* of a set of sequences Φ, written $C(\Phi)$, is the set of all pairs $<x, y>$ such that a sequence $<x, \ldots, y>$ appears in Φ. For example, if Φ contains $<+fred, +penguin, +bird>$, then $<+fred, +bird>$ is in $C(\Phi)$.

A set of sequences Φ *contradicts* the sequence $<x_1, \ldots, x_n>$ iff $<x_1, x'_i> \in C(\Phi)$ for some i, $1 \le i \le n$. Notice that contradiction occurs only in an inheritance hierarchy with exceptions. For instance, if Φ contains $<+penguin, +bird, +fly>$, then Φ contradicts $<penguin, -fly>$ and vice versa.

A token y is an *intermediary* to a sequence $<x_1, \ldots, x_n>$ in Φ iff either $y = x_i$, for some i, or Φ contains a sequence $<x_1, \ldots, x_i, y_1, \ldots, y_m, x_{i+1}>$ where $y = y_j$ for some j, $1 \le j \le m$ and $1 \le i < n$. Intuitively, y is intermediary to $<x_1, \ldots, x_n>$ iff there is a path from x through y to x_n. Notice that for a token y to be intermediary to $<x_1, \ldots, x_n>$, it is not sufficient for it to appear between $<x_i, x_{i+1}>$ where $1 \le i < n$. Consider the network in Figure 2.4. Φ has to contain the sequence $<x_1, x_2, x_3, x_4>$ in order for x_3 to be intermediary to $<x_1, x_4>$ in Φ. If it contains only $<x_1, x_2, x_5>$, then x_3 would not be so. Inferential distance ordering can now be formally described by the notion of preclusion: Φ *precludes* a sequence $\sigma = <x_1, \ldots, x_n>$ iff Φ contains a sequence $<y, x'_n>$

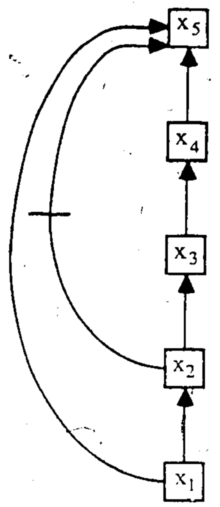


Figure 2.4 Node x_3 is intermediary to $<x_1, x_4>$

where y is an intermediary to σ in Φ. Consider the network in Figure 2.1. Φ contains $<+fred, +penguin, +bird>$ and $<+penguin, -fly>$. Φ precludes $<+fred, +penguin, +bird, +fly>$ since *penguin* is intermediary to $<+fred, +penguin, +bird, +fly>$ in Φ. We will use the terms preclusion and preemption interchangably hereafter.

In this thesis, we want to be able to say that a node is preempted by some links. Given a link $<x_1, x_n>$ in Φ, nodes $y_1, \ldots, y_n$, except x_1, are said to be *preempted* by $<x_1, x_n>$ if they are intermediary to $<x_1, x_n>$.

A sequence $\sigma = <x_1, \ldots, x_n>$ is *inheritable* in Φ iff n > 2, Φ contains both $<x_1, \ldots, x_{n-1}>$ and $<x_2, \ldots, x_n>$, and Φ neither contradicts nor precludes σ. This definition may seem strange because it requires overlapping between sequences rather than just concatenation. The reason has to do with coupling which is the property that a subclass is always in agreement with its superclasses (when there are no explicit exceptions). Suppose we define inheritability according to concatenation of sequences, then given a network shown in Figure 2.5, there would be four different grounded expansions (expansion is similar to extension and it will be formally defined shortly).

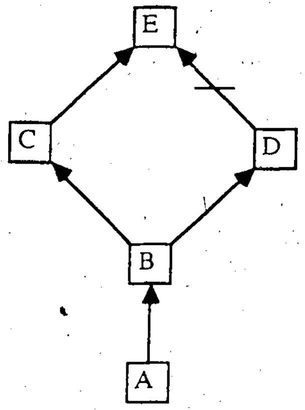


Figure 2.5 Network showing the effect of coupling

The difference among these expansions is that they contain the following sequences:

Φ_1: $<+B, +C, +E>$, $<+A, +B, +C, +E>$

Φ_2: $<+B, +C, +E>$, $<+A, +B, +D, -E>$

Φ_3: $<+B, +D, -E>$, $<+A, +B, +C, +E>$.

Φ_4: $<+B, +D, -E>$, $<+A, +B, +D, -E>$

Notice that in Φ_2, B is an E but A is not an E and in Φ_3, B is not an E while A is an E, even though in both cases, A is a B. That is, although A is a subclass of B and that there is no explicit exception, A does not inherit all the properties of B. However, if we use our definition, Φ_2 and Φ_3 are eliminated.

We say a link $<x, y_n>$ is *redundant* if the path $<x, y_1, \ldots, y_{n-1}, y_n>$ is inheritable in Φ.

Using the notion of inheritability, we can now define the superclass-subclass relations between classes and between classes and individuals: y is a *superclass* of x in Φ if $<x,y>$ is in Φ or $<x,\ldots,y>$ is inheritable in Φ. x is said to be a *subclass* of y.

Finally, some definitions to define ambiguous networks are needed. Φ is *closed under inheritance* iff Φ contains every sequence inheritable in Φ. Φ is an *expansion* of a set $S \subseteq \Sigma$ iff $\Phi \supseteq S$

and Φ is closed under inheritance. Φ is *grounded* in a set $S \subseteq \Sigma$ iff every sequence in $\Phi - S$ is inheritable in Φ. A set of ordered pairs of tokens Γ is *ambiguous* iff it has more than one grounded expansion. Γ is *unambiguous* iff it is not ambiguous.

The above definition of ambiguity is general in the sense that it does not tell us how to detect ambiguities for a given network. The following definition defines the notion of stability of an individual expansion and it is proven that a grounded expansion of a totally acyclic Γ is stable iff Γ is unambiguous. An expansion Φ is *stable* iff it contains a set of sequences of form $<x, y_1, \ldots, y_n>$, $<x, z_1, \ldots, z_m>$, $<y_1, \ldots, y_n, w>$, and $<z_1, \ldots, z_m, w'>$, such that Φ precludes neither $<x, y_1, \ldots, y_n, w>$ or $<x, z_1, \ldots, z_m, w'>$. Φ is *stable* iff it is not unstable.

Touretzky's work here is the first formalism that seeks to clearly define the inheritance principle. The inheritance principle is captured in the inferential distance ordering. Recall that in the Default Logic formalism, the minimum correctness criteria for an inheritance reasoner is that the facts returned by it have to lie within a single extension of the default theory that corresponds to the inheritance network. [Etherington 87b] shows that the reasoner described in this section does meet the requirement.

Another goal in Touretzky's work is to provide a precise semantics for the NETL representation language. The Parallel Marker Propagation Machine which implements this language is tremendously efficient. Ironically, after specifying the semantics for the system, it is found that efficient algorithm on the PMPM may not produce the correct answer. The inferential distance ordering seems to defy parallel implementation. We will have more to say about this in the next chapter.

2.7. Nonmonotonic Rules for Inheritance Systems

In the last section, we saw how the path construction technique is used to describe the inheritance principle. Although the approach overcomes the difficulties of the Default Logic and Nonmonotonic Logic formalism, it has also been criticized for its own problem [Sandewall 86]. By staying away from logical systems and dealing only with path construction, it is difficult to combine the system in [Touretzky 86] with other logic. The approach taken here by Sandewall is to incorporate the inheritance principle into the inference rules of a nonmonotonic system. The result is that a more general inference machinery is produced without sacrificing the inheritance principle.

The network in this system is specified by a set of propositions, Γ, of the form $isax(x, y, s)$ and $isa(x, y, s)$ where x and y are nodes and s is a sign, either + or −. The distinction between *isax* and *isa* is subtle. *isax* is stronger than *isa* in the sense that *isax* is *isa* except that it is independent of other links in the network. *isax* then is used for specifying explicitly known facts or facts that are independent of other links. Two other relations, *precl* and *cntr*, are used to describe the inference rules. Computing extensions in this approach is seen as adding propositions to the given set of Γ. The inference algorithm resembles Etherington's as described in Section 2.2 in the sense that an extension is generated through a sequence of steps. At step $i+1$, an inference rule is chosen which is applicable to Γ_i, and Γ_{i+1} is the union of Γ_i and the consequence of the inference rule. The difference is that contradictions may present in some extensions. These extensions are not accepted since they represent those that are not preferred. We first present the set of inference rules and then we will discuss how they work.

(1) if $isa(x, y, s)$ is a member of Γ

then add to Γ: $isa(x, y, s)$

(2) if $isa(x, y, s)$ is a member of Γ

then add to Γ: $\sim isa(x, y, -s)$

(3) if $isa(x, y, +)$ and $isa(y, z, s)$ are members of Γ

and $isa(x, z, -s)$ and $cntr(x, y, z, s)$ are not members of Γ

then add to Γ: $isa(x, z, s)$, $precl(x, y, z, s)$ and $\sim cntr(x, y, z, s)$

(4) if $precl(x, y, z, s)$, $isa(x, v, +)$ and $isa(v, y, +)$ are members of Γ

then add to Γ: $precl(x, v, z, s)$

(5) if $precl(x, y, z, s)$ and $isa(y, z, -s)$ are members of Γ

then add to Γ: $\sim isa(y, z, -s)$

(6) if $isa(x, y, +)$, $isa(y, z, +)$, $isa(x, z, -)$ and $isa(z, v, s)$ are members of Γ and $isax(y, v, s)$ is not a member of Γ

then add to Γ: $cntr(x, y, v, s)$

The purposes of rule (1) and (2) are obvious. For the rest of the rules, the propositions *precl* and *cntr* are involved. The proposition $precl(x, y, z, s)$ can be viewed as saying something like this: if $precl(x, y, z, s)$ to be true, then we can consider $is(x, z, s)$ is true but keep in mind that it may be precluded by $isa(y, z, -s)$. For $cntr(x, y, z, s)$, we may think of it as saying that it is contradictory to derive $isa(x, z, s)$. With these interpretations in mind, it is easy to see that rule (3) is used for checking if a derived fact is preempted by some "intermedi[illegible]" links. If not, then the derived fact is added to the extension. Rule (4[illegible] used to move the intermediary nodes along so that rule (3) can be used to check all the intermediary nodes. Rule (5) is used to generate a contradiction. This situation occurs when the inference rules are applied in such a way that the more general solution is derived first and when we derive the more specific solution which is in conflict with the more general solution. When a contradiction occurs, we ignore that extension. However, if the specific solution is derived first, the inhibiting conditions of rule (3) will prevent its application and hence leaving only one extension. Finally rule (6) is used to handle the case where a $isa(x, z, +)$ can be derived through $isa(x, y, +)$ and $isa(y, z, +)$ but is preempted by a given fact $isa(x, z, -)$. When this happens, we do not want to derive $isa(z, v, s)$ even if $isa(z, v, s)$ is in the original network.

2.8. Summary

In this chapter, we have provided a survey on the different formalisms for inheritance reasoning. Many of these formalisms are originally intented for formalizing common sense reasoning. These formalisms are more general and powerful. Nevertheless, they have problems in capturing the inheritance principle implicitly.

The usefulness of inheritance systems and the inability of those formalisms to capture the inheritance principle motivated Touretzky to study inheritance systems independently. His mathematical formalism provides a precise formalization of the inheritance principle. Recently, Etherington shows that the inferential distance ordering satisfies the minimal correctness requirement of the default logic formalism as discussed in Section 2.2 [Etherington 87b]. That is, the results generated by an inferential distance inheritance reasoner lie within a single extension of the default theory that corresponds to the given inheritance network. Etherington also observes that a lot of the results (i.e. theorems and the mechanism for determining extension) in [Touretzky 86] have close correspondence to that of the default logic formalism. Given that the inferential distance ordering satisfies the minimal correctness requirement, Etherington argues that the inferential distance ordering "can be viewed as fast inference algorithms for reasoning with the tractable class of default theories that correspond to inheritance networks." The correspondence between the two formalism need to be further explored.

In Section 2.4, we discussed Poole's theory comparator. He shows that inheritance reasoning can be done using the theory comparator. The system described is more general since it allows arbitrary well-formed formulae in the defaults. Reasoning in a semantic network is seen as a restricted form of reasoning using the theory comparator. This kind of restricted reasoning can be supported by a parallel machine, the Parallel Marker Propagation Machine, which will be discussed in further detail in the next chapter. In addition, Touretzky's formalism is catered to such form of reasoning and allows us to concentrate on the problems of inheritance reasoning in a semantic network. The later investigations of this thesis is therefore based on Touretzky's formalism. In Chapter 4 and 5, we present two algorithms which reason according to the Touretzky's definition.

Chapter 3

Parallel Inheritance Algorithms

3.1. Introduction

We have briefly introduced the Parallel Marker Propagation Machine (PMPM) in Chapter 1. In this chapter, we discuss in more detail the machine and a language that allows us to write algorithms that run on the PMPM conveniently. We then give a survey of a number of parallel or pseudo-parallel inheritance algorithms. In view of the complexity of inheritance reasoning, a total parallel algorithm seems impossible. In this chapter, we see how we can make trade-offs between generality of networks and parallelism; and between correctness and parallelism.

3.2. The Parallel Marker Propagation Machine

The aim of the PMPM is to provide fast inference given a large database of real-world knowledge. The idea behind it is straightforward: the nodes and links of a semantic net are directly mapped into hardware processing elements and hardware links respectively. Since the aim is to capture a sufficiently large amount of knowledge, the PMPM is expected to have millions of nodes. This means that the hardware processing elements have to be very simple; otherwise, it would be practically impossible to produce such a machine.

In the PMPM, each node is very simple. Each of them has a unique id-number for addressing purposes, a number of state bits and a bit wide boolean logic unit. The state bits can be set or cleared by the commands issued by a central controller. The central controller can broadcast commands to either an individually addressed node, or to a set of nodes based on their internal state bits. Inference is done by propagating marker bits around the network formed by the nodes and links. The result of the inference is reflected by the marker bits that the nodes received during the propagation. The PMPM is an SIMD machine. At each time step, multiple nodes can propagate markers to different nodes and hence parallelism is achieved.

3.2.1. A PMPM Language

[Touretzky 86] develops a language for marker propagation algorithms. The language is simple and easy to understand. In Chapter 4 and 5, we will present two algorithms that are written in this language. It is important for the reader to understand the material in this section. Readers who are familiar with the language may skip the section without loss of continuity.

As we have mentioned, the PMPM is an SIMD machine. Both the processing elements and the links respond to commands from the central control. There are two sets of commands: node commands and link commands. The marker bits of a node can be set and cleared by the following two node commands:

`set` $[M_1, \ldots, M_n]$

`clear` $[M_1, \ldots, M_n]$

where M_i's are markers. Or, they can be set and cleared by the following link commands:

`set_head` $[M_1, \ldots, M_n]$

`set_tail` $[M_1, \ldots, M_n]$

`clear_head` $[M_1, \ldots, M_n]$

`clear_tail` $[M_1, \ldots, M_n]$

Given a link $A \rightarrow B$, A is said to be the tail node and B the head node of the link respectively. The above commands are unconditional statements. Conditional statements are of the form:

$condition_1, \ldots, condition_n \Rightarrow action_1, \ldots, action_m$

If $condition_1$ through $condition_n$ are satisfied, $action_1$ through $action_m$ are performed. Conditions here refer to marker bits that are on or off at a node such as:

`on` $[M_1, \ldots, M_n]$

`off` $[M_1, \ldots, M_n]$

`any_on` $[M_1, \ldots, M_n]$

`any_off` $[M_1, \ldots, M_n]$

The "on" ("off") condition is satisfied if all of the markers mentioned are on (off). The "any_on" ("any_off") condition is satisfied if any of the markers mentioned are on (off). Another condition is that a unique name be assigned to a node:

```
name[x]
```

The above are tests that are performed by a node. Tests can also be performed by links on their head nodes and tail nodes:

```
on_head[M1,...,Mn]
off_head[M1,...,Mn]
any_on_head[M1,...,Mn]
any_off_head[M1,...,Mn]
on_tail[M1,...,Mn]
off_tail[M1,...,Mn]
any_on_tail[M1,...,Mn]
any_off_tail[M1,...,Mn]
```

Finally, a link can test if it is of a specific type:

```
link_type[t1,...,tn]
```

where t_i's are different types of links. Another important construct in the PMPM language allows us to perform iteration:

```
loop
    body
endloop
```

When the loop is entered, each statement in body is executed. If none of the statements inside body can be executed, i.e. the conditions of these statements are not met, the loop exits. If at least one statment inside body is executed, the loop is repeated.

To illustrate the use of the language, consider the procedure shown in Figure 3.1. It is used to compute the transitive closure of the IS-A link from a starting node x. At line 3, all nodes respond to this node command and clear their respective marker M_{TC}. At line 4, only the given node x sets its marker bit M_{TC} on. At each step inside the loop, all the nodes that have an IS-A link as an input link and that the tail node of the link is already marked with M_{TC} are marked by M_{TC}. When no more nodes can be marked, the loop terminates and the procedure exits. The nodes marked with M_{TC} is the transitive closure of node x. The run time of the procedure is proportional to the depth of the network and does not depend on the actual number of nodes that the network has.

```
1  Procedure transitive_closure (x: node)
2  Begin
3     clear[$M_{TC}$];
4     name[x] ⇒ set[$M_{TC}$];
5     loop
6       link_type["→"], on_tail[$M_{TC}$], off_head[$M_{TC}$] ⇒ set_head[$M_{TC}$];
7     endloop
8  End.
```

Figure 3.1 Transitive closure algorithm

3.3. The Upscan Algorithm

[Touretzky 86] presents a parallel algorithm called *upscan* which runs on the PMPM. The algorithm is basically a shortest path algorithm which may produce erroneous results. Two methods are proposed to ensure the correct results are obtained. First, let the algorithm run on a restricted class of inheritance networks. By imposing certain restrictions on inheritance networks, it is possible to guarantee the correctness of the shortest path algorithm. A class of inheritance networks that the shortest path algorithm can deal with properly are called orthogonal class/property inheritance networks. The second method involves modifying the inheritance network so that the *upscan* algorithm works correctly. Such a modification is called conditioning the network.

3.3.1. Orthogonal Class/Property Multiple Inheritance Networks

As has been mentioned, the shortest path algorithm is inadequate to deal with general inheritance networks. The aim here is to characterize a restricted class of inheritance networks that will not cause problems for the shortest path algorithm. Due to the simplicity of the shortest path algorithm, one might expect that the restriction would be quite severe. We first describe a class of networks which distinguishes class nodes and property nodes. We then discuss a class of networks which does not allow a node to inherit the same properties from multiple superclasses. The first class is called class/property multiple inheritance networks. The second is called orthogonal class/property multiple inheritance networks.

In the most general treatment of inheritance networks, each node can represent either a class of objects or some properties of an object. In a class/property inheritance network, we distinguish nodes which represent classes and nodes which represent properties. In such a network, only the inheritance of properties can have exceptions. The inheritance of class membership is strictly indefeasible. Figure 3.2 shows a class/property inheritance network. Nodes P_1 and P_2 represent properties while nodes C_1 through C_9 represent classes.

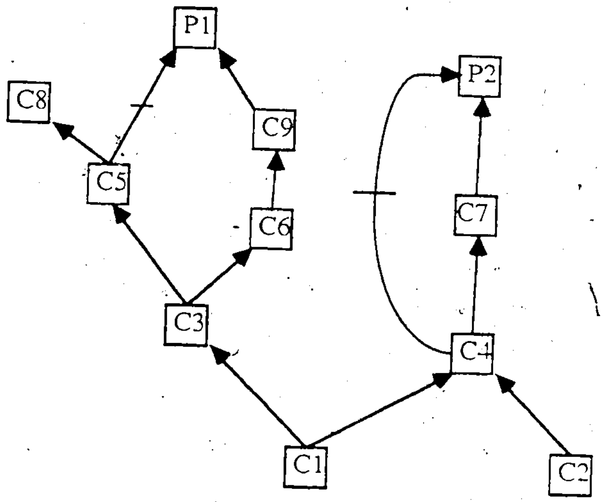


Figure 3.2 A class/property inheritance network

Class/property inheritance networks are still too complicated for the shortest path algorithm to work correctly. The reason is that a class/property inheritance network can be ambiguous. For instance, the subnet formed by nodes C_3, C_5, C_6, C_9 and P_1 in Figure 3.2 is an ambiguous network. An orthogonal class/inheritance inheritance network further restricts general inheritance system such that, if a class inherits properties from multiple superclasses, these properties have to be disjoint. More precisely, suppose there are links $A \rightarrow B$ and $A \rightarrow C$. Then for any node D, either there is a path from B to D or from C to D but not both. Figure 3.3 shows an orthogonal class/property inheritance network. Notice that the network in Figure 3.2 is not orthogonal. That is because there are links $C_3 \rightarrow C_5$ and $C_3 \rightarrow C_6$, and we can go from both C_5 and C_6 to P_1.

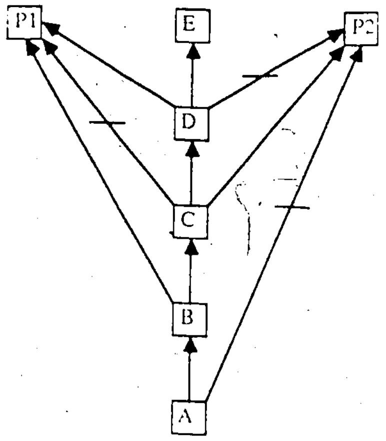


Figure 3.3 An orthogonal class/property inheritance network

Touretzky shows that an orthogonal class/property inheritance network is always unambiguous. He also shows that the *upscan* algorithm works correctly given an orthogonal class/property inheritance network.

3.3.2. Conditioning

The second method that is used to help the *upscan* algorithm is called conditioning. The idea is to modify the given inheritance network so that the shortest path algorithm can work correctly. In [Touretzky 86], an additive approach is taken, i.e. to add new links to the network instead of restructuring the network. Figure 3.4 shows a network that beats the shortest path algorithm. Figure 3.5 shows an additively conditioned version of the network in Figure 3.4.

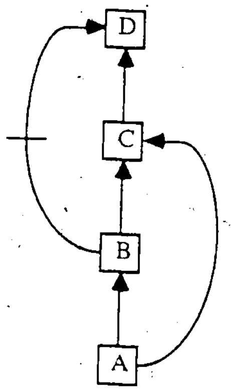


Figure 3.4 A network that beats the shortest path algorithm

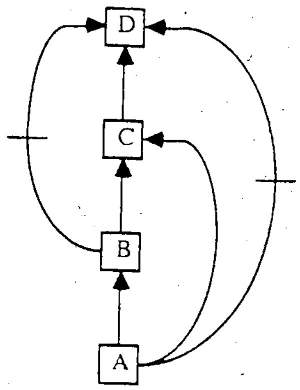


Figure 3.5 An additively conditioned network

Link $A \not\rightarrow D$ is added to the original network. Notice that the shortest path algorithm will conclude that A is not a D which is what we want.

The conditioning algorithm will add the right link only if it knows what the correct answer is. The implication is that we have to compute the preferred extension(s) of the given network first and then let the conditioning algorithm add the appropriate links. Computing the preferred extension(s) and the conditioning is done by a Maclisp program called TINA (Topological Inheritance Architecture). Instead of simulating marker propagation in a PMPM, lists of possible marker propagation paths are constructed and analyzed. This ensures that the inheritance ordering is preserved by adding links which block problematic paths.

It is possible to condition any consistent inheritance network using the additive approach. The simplest way is to connect each node in the network to every other with an appropriate link. The result can then be obtained in one parallel step. The problem, however, is that we have a network with $O(N^2)$ links. [Touretzky 86] shows a more efficient conditioning algorithm. But in the worst case, we still cannot have a network with $O(N^2)$ links.

3.4. Partitioning and Quasi-Parallel Inheritance Algorithm

The *upscan* algorithm described in the last section is extremely efficient: it is completely parallel. Unfortunately, its great efficiency is also its downfall. The approach taken here is to trade off some parallelism for correctness. If a reasoner produces wrong answers, it does not matter how fast it can reason. Therefore this trade off between parallelism and correctness is another reasonable approach for dealing with parallel processing of inheritance networks.

[Etherington 87a] suggests a method for inheritance reasoning in which limited parallelism can be achieved. Recall that in the Default Logic approach to formalizing inheritance reasoning, exceptions are explicitly stated. We can view the problem with the shortest path algorithm as the problem of dealing with these exceptions. Without exception, we can process networks in parallel. The idea of this method is to partition the given network into subnets which can be processed in parallel. A mple (Figure 3.6) is given to show how the method can be used to construct extension in quasi-parallel.

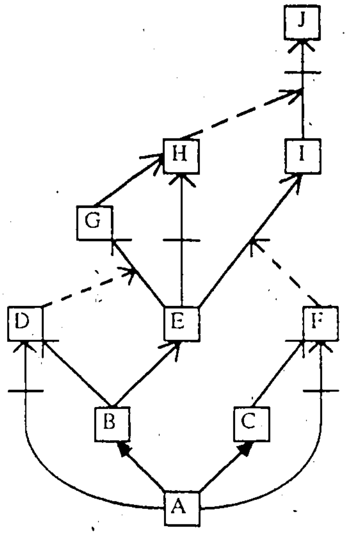


Figure 3.6 Example for quasi-parallel algorithm

We start by numbering each node in the network according to the number of exception links upon which it depends. For example, nodes A, B, C, D, E and F do not depend on any exception link. They are assigned as level 1. Node H is a special node in this network. If we consider the path $E \rightarrow G \rightarrow H$, then node H depends on one exception link. However, if we consider the path $E \not\rightarrow H$, the node does not depend on any exception link. When a node can be assigned to different levels, we pick the smallest level number for it. So in this case, node H is assigned as level 1. Nodes G and I depend on one exception link and are assigned as level 2. Finally node J depends on two exception links and is assigned as level 3. If the nodes are assigned to k different levels, we perform k parallel steps. At each step n, all the links having exceptions at the last step are ignored. We then process the remaining network in parallel by propagating markers from those nodes with level number $n - 1$. Etherington shows that after k steps, we obtain an extension. Notice that links [illegible] $\rightarrow B$ and A $\rightarrow$ [illegible] re strict links. It is assumed that propagation through strict links is instantaneou

Here, we have no[illegible]cified any specific marker propagation scheme. Different schemes may lead to different extensions. For example, in Figure 3.6, we may have four different extensions depending on the marker propagation scheme:

$E_0 = \{A, B, C, E, \sim H, D, F\}$

$E_1 = \{A, B, C, E, \sim H, D, \sim F, I, \sim J\}$

$E_2 = \{A, B, C, E, \sim H, \sim D, F, G\}$

$E_3 = \{A, B, C, E, \sim H, \sim D, \sim F, I, G, \sim J\}$

The method described above is correct in the sense that the conclusions drawn lie within a single extension. Nonetheless, it is not complete since some extensions may never be generated no matter what marker propagation scheme you use. For example, the following extensions may never be generated:

$E_4 = \{A, B, C, E, H, \sim D, F, G\}$

$E_5 = \{A, B, C, E, H, \sim D, \sim F, G, I\}$

The bias towards certain extension depends on how we partition the network and hence depends on the exception links. In the example, if the exception link $\frac{E(x) : G(x), \sim D(x)}{G(x)}$ is removed, then extensions E_4 and E_5 can be generated. This makes it very difficult to characterize the bias of the algorithm. This is in sharp contrast with computing the preferred extension which is based on the well understood notion of inferential distance ordering.

3.5. Skeptical Reasoning in Inheritance Systems

The inheritance reasoner as defined in [Touretzky 86] is a credulous reasoner in the se

that it tries to conclude as much as possible given an inheritance network. For example, consider the Nixon Diamond in Figure 3.7.

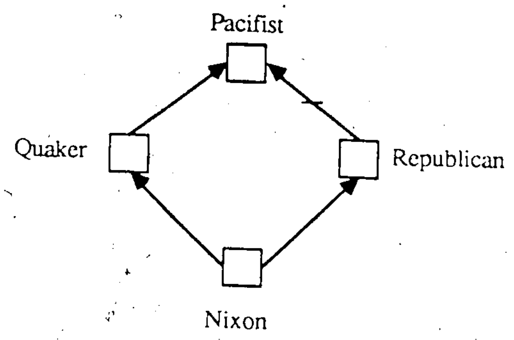


Figure 3.7 Nixon Diamond

In a credulous reasoner, two preferred extensions are generated. In one extension, we would conclude that Nixon is a Pacifist while in the other, he is not. On the other hand, a skeptical reasoner refuses to conclude anything when multiple preferred extensions are discovered. For the network in Figure 3.7, a skeptical reasoner would refuse to conclude anything about whether Nixon is a Pacifist or not. A result of this is that a skeptical reasoner will generate a unique extension while a credulous reasoner may generate multiple extensions. Since a skeptical reasoner need not consider multiple extensions, skeptical reasoning algorithm should be less complicated than credulous reasoning algorithms.

Consider for a moment the practical implementation of a credulous reasoner using the PMPM. As has been mentioned, it is difficult to use the MF 1 to compute the preferred extension of a given network. When multiple preferred extensions exist, the problem becomes even more difficult. The strategy adopted in [Touretzky 86] in designing TINA (see Section 3.1) is to let the program abort and return a message whenever multiple preferred extensions are found. At first glance, this seems to result in a reasoner that is similar to a skeptical reasoner. But the difference between concluding ambiguity and concluding nothing has a significant impact on the processing of the rest of the network. For instance, consider the network shown in Figure 3.8 (from [Touretzky et al 87]).

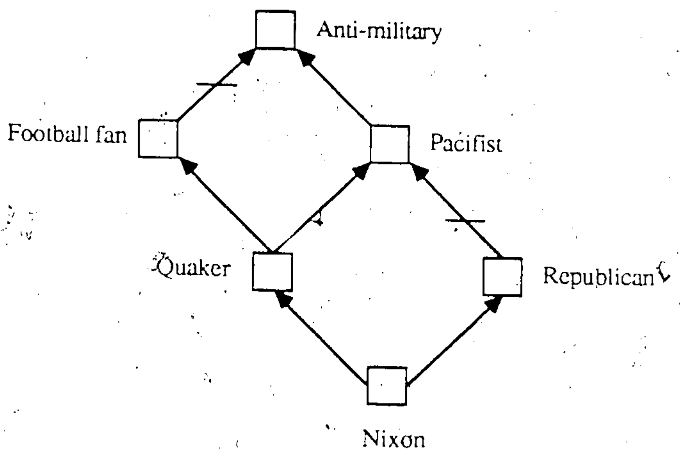


Figure 3.8 Network showing cascaded ambiguities

For a credulous reasoner, there are three preferred extensions:

$E_0 = \{Nixon, Quaker, Republican, Pacifist, Football\ fan, Anti\text{-}military\}$

$E_1 = \{Nixon, Quaker, Republican, Pacifist, Football\ fan, \sim Anti\text{-}military\}$

$E_2 = \{Nixon, Quaker, Republican, \sim Pacifist, Football\ fan, \sim Anti\text{-}military\}$

TINA will be aborted when it encounters the node *Pacifist*. This is because in E_0 and E_1, *Nixon* is a *Pacifist* while in E_2, *Nixon* is not. The skeptical reasoner in [Horty et al 87], on the other hand, simply refuses to conclude about whether *Nixon* is a *Pacifist* or not. The effect of this is that it will conclude that *Nixon* is *~Anti−military* since nothing is passed from the link *Pacifist* $\rightarrow$ *Anti−military* to oppose the input from the link *Football fan* $\not\rightarrow$ *Anti−military*.

[Horty et al 87] presents a serial-parallel algorithm running on the PMPM for skeptical inheritance reasoning. In the previous two sections, the algorithms are concerned with computing or reconstructing the whole extension for a network. But in real applications, what we want to know is whether node x is a subclass of node y. In processing such a query, we do not want the system to consider links that are irrelevant to that particular query. By restricting the attention to only relevant links, an inheritance reasoner may analyze a smaller and possibly less complex network. [Horty et al 87] specifies a restriction of a net Γ with respect to the particular kind of query. This query-restricted network $\Gamma^{x,y}$ contains all the links that are necessary to determine whether x is a y. Formally, $\Gamma^{x,y}$ is defined as the minimal net containing:

(1) every link on every path in Γ from x to y, and

(2) every link on every path in Γ from x to w, for all nodes w occurring in $\Gamma^{x,y}$.

The purpose of the procedure `trim_for_query(x,y)` is to determine $\Gamma^{x,y}$ for a particular query. After running the procedure shown in Figure 3.9, each node w in Γ will be marked with M_2 iff $w = x$ and w occurs in $\Gamma^{x,y}$. In subsequent processing, we only need to restrict attention to those nodes that are marked with M_2.

```
Procedure trim_for_query(x,y: node)
Begin
 clear[$M_x$, $M_y$ , $M_1$ , $M_2$];
 name[x] ⇒ set[$M_x$];
 name[y] ⇒ set[$M_y$];
 loop
  link_type["→"], any_on_tail[$M_x$, $M_1$], off_head[$M_1$]
     ⇒ set_head[$M_1$];
 endloop;
 link_type["→","↛"], on_tail[$M_1$], on_head[$M_y$] ⇒ set_head[$M_2$];
 loop
  link_type["→","↛"], on_head[$M_2$], on_tail[$M_1$], off_tail[$M_2$]
     ⇒ set_tail[$M_2$];
 endloop
End.
```

Figure 3.9 Procedure trim_for_query(x, y)

[Horty et al 87] introduces the idea of the $Degree^{x,y}$ of a path. Basically it refers to the length of the longest path between node x and node y. At step n, the serial-parallel algorithm will consider all the nodes with degree n. When there are nodes that have conflicting inputs, the algorithm considers them one by one, i.e. serially. The algorithm tries to determine if any of the immediate links to a node dominate. This is done by propagating a marker from all the immediate links. Those immediate links that receive the marker after the propagation are preempted. If only one is left as unpreempted, then the node in question is marked by the marker that is passed from that link. If more than one such link is found and they are of different types, then the node in question is ambiguous. If there are no conflicting inputs at step n, markers are propagated in parallel to nodes at degree $n+1$. So as we can see, parallel processing is used whenever it is possible but occasionally, serial processing is used to check for ambiguity.

3.6. Summary

In this chapter, we have discussed a number of parallel algorithms for inheritance reasoning. We can summarize these as follows:

(1) Restrict the class of inheritance networks so that the simple shortest path algorithm is adequate.

(2) Modify the given inheritance network so that the simple shortest path algorithm can work properly.

(3) Sacrifice completeness to provide a quasi-parallel algorithm.

(4) Redefine and simplify inheritance reasoning so that a serial-parallel algorithm can be designed.

Due to the simplicity of the shortest path algorithm, the first approach results in a very restricted class of inheritance networks: orthogonal class/property inheritance networks. Besides theoretical interest, it is doubtful that this class of inheritance networks can be practically useful. The problem with the second approach is that the conditioning process cannot be done in parallel and reconditioning has to be done every time the network is being updated. In addition, using the additive approach to conditioning, the number of links added is $O(N^2)$ in the worst case, where N is the total number of nodes in the network. Other kinds of conditioning methods have not been investigated. For instance, restructuring the network may lead to a simpler network. However such a conditioning algorithm might be complicated since it involves both deleting unnecessary links and adding useful connections. In the third approach, Etherington abandons the idea of having a totally parallel inheritance algorithm due to the inadequacy of the simple shortest path algorithm and the complexity of the inheritance reasoning. He observes that although we cannot process the entire network in parallel, it is possible to process subnets in parallel by ignoring exception links. It turns out that we have to sacrifice more than just some of the parallelism in order to get a correct algorithm. The quasi-parallel algorithm is incomplete and it is difficult to characterize how the algorithm favors which extensions. In the fourth approach, Horty and his colleagues argue that the skeptical theory of inheritance reasoning is equally well-motivated as the credulous theory. It has another advantage: due to its relative simplicity, simpler and more efficient inheritance algorithms for skeptical reasoning can be designed. However, the question of whether we can have a parallel or even a serial-parallel algorithm for inheritance reasoning as originally defined in [Touretzky 86] is still left unanswered.

Chapter 4

Dealing with Unambiguous Inheritance Networks

4.1. Introduction

The PMPM is originally proposed as a parallel machine for Artificial Intelligence. A number of useful inference such as set intersection, transitive closure and property inheritance can be done in parallel using such a machine. Unfortunately, due to the complexity of nonmonotonic multiple inheritance systems, simple algorithms running on the PMPM is inadequate for the job. In the last chapter, we saw a number of attempts to circumvent the situation such as preprocessing the given network or limiting the application of the shortest path algorithm to some simple class of networks, etc. In this chapter, we consider another approach. Since part of the reason why the PMPM is not able to deal with nonmonotonic multiple inheritance systems is its oversimplicity, it is reasonable to investigate a possible enhancement of the PMPM such that it is more suitable for the job. The idea of the approach taken here is to let each node pass additional information besides markers to other nodes. We allow nodes to pass marker-value pairs instead of just simple markers and also allow them to do some simple analysis from these marker-value pairs. The result is that we can have a totally parallel algorithm for unambiguous networks running on such a machine. We also provide an argument for the correctness of the algorithm according to the definition of inheritance in [Touretzky 86].

4.2. Unambiguous Inheritance Networks

The class of unambiguous inheritance networks is a major class of inheritance networks which includes the class of orthogonal class/property inheritance networks. The networks in this class have one extension. To be exact, these networks have only one preferred extension. The preferred extension is determined by the inferential distance ordering as defined in [Touretzky 86]. From now on, extension refers to preferred extension unless explicitly stated otherwise. The class of unambiguous inheritance networks is useful since most of our knowledge is unambiguous, i.e. we can determine whether an entity is a subclass of another.

Since there is only one preferred extension in an unambiguous inheritance network, algorithms dealing with only unambiguous inheritance networks should be simpler. One extreme is the class of orthogonal class/property inheritance networks. The class is so restrictive that the simple shortest path algorithm running on the PMPM is sufficiently powerful. Nevertheless, orthogonal class/property inheritance networks are too simple. General unambiguous networks are more useful and interesting. A completely parallel inheritance algorithm for unambiguous networks is desirable.

4.3. Problem with Unambiguous Networks

Recall that there are two problems associated with the simple shortest path algorithm in dealing with general inheritance networks. The first results when redundant links are present in the network. The second when ambiguity is involved. When there are redundant links, the shortest path algorithm may produce incorrect answers. When ambiguity is involved, the algorithm would bias towards certain extensions. Since in this chapter we restrict our attention to only unambiguous networks, we do not need to worry about the second problem. .

Computationally, the problem with redundant links is that they provide shortcuts in the network. Through these shortcuts, some markers can "jump the line" and cause trouble. We need to restrict this kind of problematic activity. We can also look at the problem in another way. We can view the marker propagation as a race between the markers to reach a node. For instance, in Figure 4.1, starting from node A, an M_F can reach node C in one step while an M_T marker can reach node C in two steps. Node C is therefore marked as M_F.

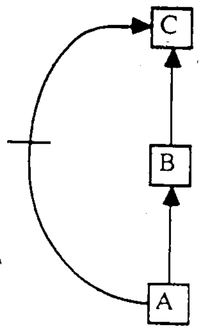


Figure 4.1 Network showing race condition: nodes are marked correctly

In Figure 4.2, an M_T marker reaches node E in two steps while it takes three steps for an M_F marker to reach E. Node E is incorrectly marked as M_T. That is, we have a race condition between the markers. We need to control this race condition in order to solve the problem.

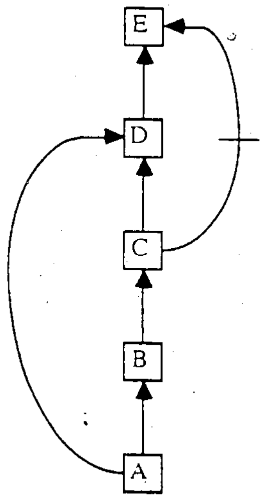


Figure 4.2 Network showing race condition: nodes are marked incorrectly

4.4. Enhancing the PMPM

The propagation of markers can be seen as the firing of nodes. For instance, in Figure 4.2, the firing of node A results in the propagation of an M_T marker to both node B and node D. Therefore, controlling the race condition between markers is equivalent to synchronizing the firing of nodes. We do not allow a node to fire until all its inputs have arrived. We can modify the machine so that each processing element will not propagate a marker until all of its inputs have arrived. But what we are looking for is a minimum change in the PMPM. If the synchronization can be done by a software method and that it is not too expensive to implement, then we prefer an algorithm which can do the job. [Horty et al 87] presents an algorithm called `select_next_degree` which is adequate for our purpose. Since we want the algorithm to select all those nodes that can fire, we will call the algorithm `select_node`. The procedure is used in conjunction with `trim_for_query(x,y)` which markes all the nodes in the query-restricted network $\Gamma^{x,y}$ with the M_2 marker (see Section 3.5).

```
1 Procedure select_node()
2 Begin
3   clear[$M_{Fire}$];
4   link_type[$\rightarrow$, $\not\rightarrow$], any_on_tail[$M_x, M_{old}$], on_head[$M_2$],
5       off_head[$M_{old}$] $\Rightarrow$ set_head[$M_{Fire}$];
6   link_type[$\rightarrow$, $\not\rightarrow$], on_head[$M_{Fire}$], on_tail[$M_2$],
7       off_tail[$M_x, M_{old}$] $\Rightarrow$ clear_head[$M_{Fire}$];
8   on[$M_{Fire}$] $\Rightarrow$ set[$M_{old}$];
9 End.
```

Figure 4.3 Procedure select_node()

Basically, we want the algorithm to mark all the nodes that can fire by the marker M_{Fire} and all those that have already been fired by the marker M_{old}. At each call to the procedure, all nodes with at least one child node fired during the previous step are marked with M_{Fire}. This is done at lines 4 and 5. But if one of the child nodes is not fired during the last step, the marker is cleared.

This is done at line 6 and 7. Finally, the statement in line 8 sets all the nodes that are ready to fire with marker M_{old} preparing for the next iteration.

When the firing of each node is not controlled, each node will be marked by the marker that arrives first. But now we control each node such that it will wait for all of its input. The problem is how it should be marked when conflicting input arrives. For example, in Figure 4.2, node E would receive both an M_T and M_F. It has to choose between one of them. The decision has to be made using some additional information. It turns out that an additional number is adequate for the nodes to make the correct decision. Instead of passing just markers, each node now passes a marker-value pair to the ancestor. Hence, the enhanced PMPM is like a combination of marker-passing and value-passing machine [Fahlman, Sejnowski 83]. The resulting machine is similar to Thistle except that here we explicitly try to use such a hybrid machine to perform correct inheritance reasoning. Since each node passes a marker-value pair, we need to modify the node and link commands which set the marker of a node so that they now set the marker and the value of a node. In addition, there are two more node commands: `max_level()` and `min_level_marker`. The `max_level` command finds the maximum level number among the child nodes. The `min_level_marker` command finds the marker of the child who has the smallest level number among the children with marker M_T and M_F.

4.5. A Parallel Algorithm for Unambiguous Inheritance Networks

In this section, we present a parallel algorithm which runs on the enhanced PMPM. We will assume that the question posed for the algorithm is of the form: "tell me if x is a subclass of y."

When the procedure `subclass(x,y)` is called, node x is marked with an initial level number 1 and an M_T marker at line 4. Then each node in the network repeats the following until no node can be fired. Each node checks if all its children have provided information. If any one of its children has not arrived yet, it does nothing. In the procedure, this is done by calling

```
Procedure subclass(x,y: node)
Begin
  trim_for_query(x,y);
  name[x] $\Rightarrow$ set[$M_T$, #1];
  loop
    select_nodes();
    on[$M_{Fire}$] $\Rightarrow$ set_level[max_level() + 1], set[min_level_marker()];
    link_type[$\rightarrow$], on_tail[$M_{Fire}$, $M_T$] $\Rightarrow$ set_head[$M_T$, level];
    link_type[$\not\rightarrow$], on_tail[$M_{Fire}$, $M_T$] $\Rightarrow$ set_head[$M_F$, level];
    link_type[$\rightarrow$, $\not\rightarrow$], on_tail[$M_{Fire}$, $M_F$] $\Rightarrow$ set_head[$M_D$, level];
  endloop
End
```

Figure 4.4 A parallel algorithm for unambiguous inheritance networks

select_node(). select_node() marks all the nodes whose inputs have arrived with an M_{Fire} marker. When all of the children of the node have arrived, the node has to propagate a level number and a marker to the next level. The level number to be passed is equal to one plus the maximum of the level numbers of the children. To decide what marker to propagate, it finds, among the children with either M_T or M_F, the one with the smallest level number. It can be shown that if the child is not unique, i.e., if there is more than one child having the same lowest level number, then the markers of these child nodes must be the same; otherwise, the network is ambiguous (see Section 4.7). If there is one, then it is marked with that marker. When the propagation is done, if node y is marked with M_T, then x is a subclass of y, otherwise it is not.

4.6. Examples

This section presents some examples showing how the algorithm works on an inheritance network. In the following diagrams, the rectangular nodes represent processing elements in the enhanced PMPM. In terms of an inheritance hierarchy, each node represents either an individual or a class. The marker-value pairs and markers inside the node indicate that the markers are turned on. In each of the examples below, a [M_T, #1] marker-value pair is given, at time T=1, to the node x.

Example 1 shows a simple inheritance hierarchy with exceptions. Although *royal elephant* is an *elephant*, it does not inherit the property of an *elephant*, namely being a *gray thing*. At time T=1, node *royal elephant* is given a marker-value pair [M_T, #1] and it is the only node that is marked with M_{Fire}. At T=2, node *elephant* is marked with M_{Fire} and receives [M_T, #2] from node *royal elephant*. Node *gray thing* receives [M_F, #2] from *royal elephant*, too. At the final step, *gray thing* receives [M_T, #3] from *elephant*. Since [M_F, #2] has a smaller level number, this input from *royal elephant* is chosen. This network can be handled by a shortest path algorithm as well as our algorithm.

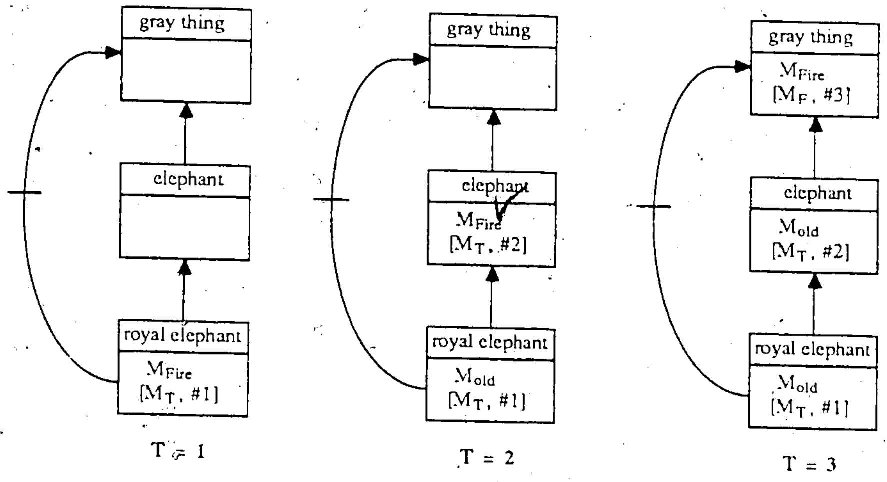


Figure 4.5 Example 1

The second example shows a more complex network. A redundant link, namely that *clyde* is an *elephant*, is present in the network. Without this link, we can infer that *clyde* is a subclass of *circus elephant*, *royal elephant* and *elephant*, and that *clyde* is not a *gray thing*. But when the redundant link is added, a shortest path algorithm will infer that *clyde* is a subclass of everything, including *gray thing*. Conditioning is required in this network for a shortest path reasoner to infer correctly [Touretzky 86]. Our algorithm, however, is able to produce the right answer without preprocessing the network. This network also shows that whether a node is a subclass of another does not depend on the length of the inference path; rather, it depends on the structure of the inheritance network. In here, to infer that *clyde* is a *gray thing* takes only two steps while inferring it not being a *gray thing* takes three steps. Nevertheless, the structure of the network demands that we choose the longer inference path.

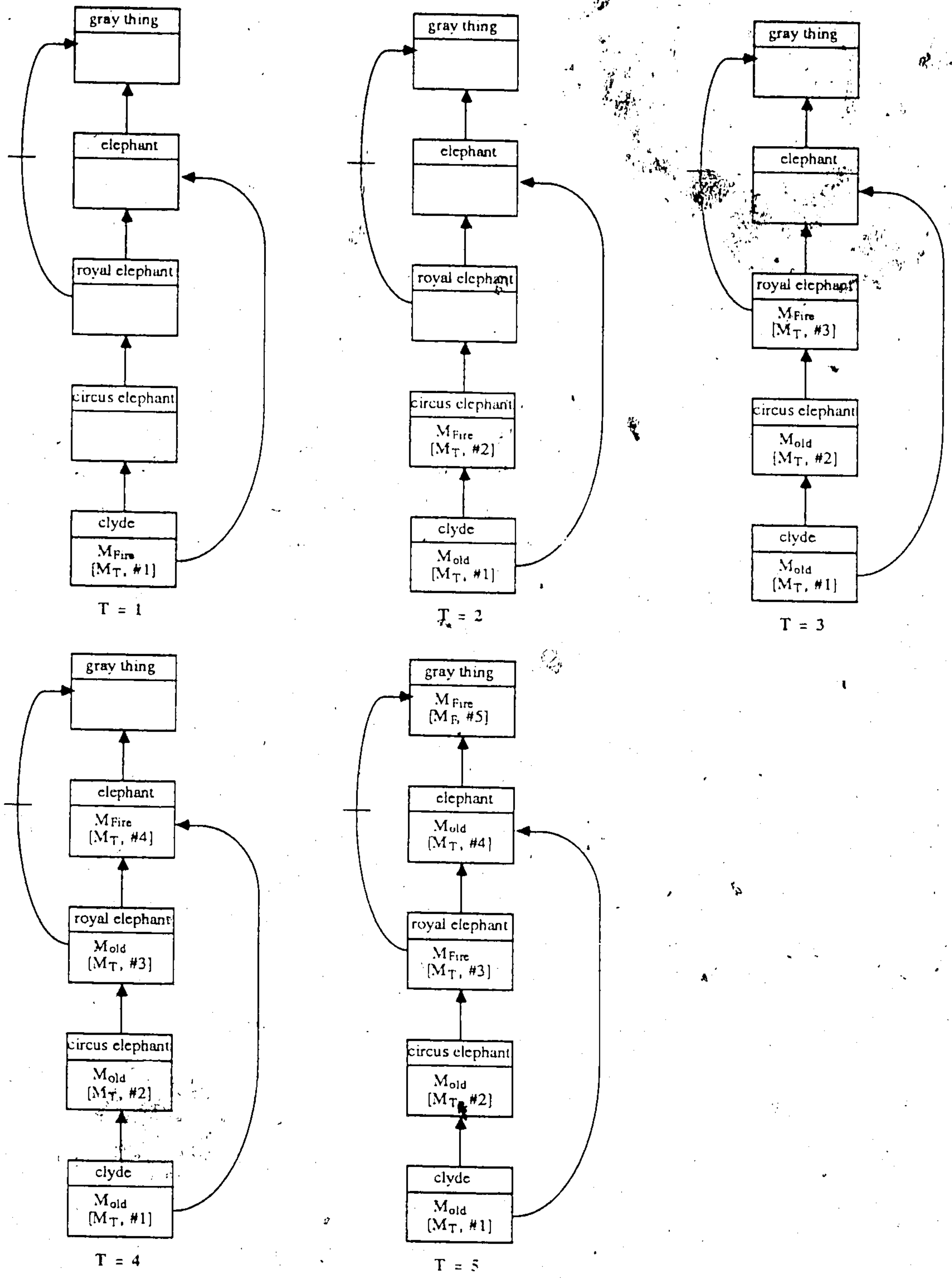


Figure 4.6 Example 2

4.7. Correctness

In this section, we provide an argument for the correctness of the algorithm. Before we do so, let us discuss the intuitive reason why the algorithm works. Recall that the inferential distance ordering allows us to resolve ambiguity of a network when possible. Basically, it is a partial order. For instance, the nodes in the network as shown in Figure 4.7 are numbered according to the inferential distance ordering.

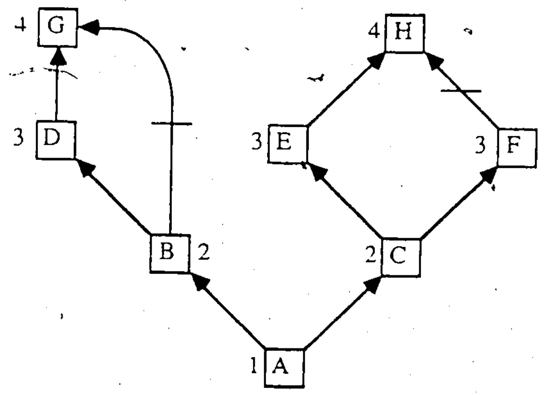


Figure 4.7 Network showing the inferential distance ordering

It is a partial order since nodes E and F are both numbered the same. Node H is ambiguous since node E does not have preference over node F and vice versa. Now when we restrict our attention to only unambiguous network, the inferential distance ordering becomes a total order. The algorithm shown above basically numbers the nodes according to the inferential distance order. When the nodes are so ordered, the node with a smaller number overrides the ones with larger number when dispute arises.

Given Γ, an unambiguous set of well-formed inheritance assertions, let x be the node of which we want to find the superclasses. Let w be an ancestor node of x, i.e., there exists a path $<x, y_1, \ldots, y_n, w>$ in Φ. Suppose there is more than one path from x to w. Let L denote the maximum path length of x, i.e., the length of the longest path from x to w. Suppose we start the execution of the algorithm at time = 1. Then according to the algorithm, a node with maximum path length L will be fired at time = L. Also, all nodes with maximum path length less than L must have been fired before time = L. This is in sharp contrast with the shortest path algorithm in which a node with a minimum path length L' will always be fired at time = L'.

Before we show the correctness of the algorithm, we show that if there is more than one child that arrives the earliest and that the markers of these child nodes are not the same, then the network is ambiguous. Notice that the child who arrives first has the lowest level number. Suppose an expansion Φ of Γ contains the sequences $<x, y_1, \ldots, y_n>$, $<x, z_1, \ldots, z_m>$, $<y_1, \ldots, y_n, w>$ and $<z_1, \ldots, z_m, w'>$ and that both y_n and z_m arrive at w at the same time. We then have $n = m$. Since $n = m$, there is no y_i, $1 \leq i \leq n$ such that y_i is intermediary to $<x, z_1, \ldots, z_m, w'>$. Otherwise, $m > n$. Similarly, there is no z_j, $1 \leq j \leq m$, such that z_j is intermediary to $<x, y_1, \ldots, y_n, w>$. Otherwise, $n > m$. Therefore Φ neither precludes $<x, y_1, \ldots, y_n, w>$ nor $<x, z_1, \ldots, z_m, w'>$; i.e., Φ is unstable. Therefore Γ is ambiguous.

Now we show the correctness by induction. The base case is trivial. Suppose at time t every node is marked correctly, and at time $t+1$, node w is fired. We show that w is correctly marked. According to the algorithm, node w will pick the input from a child, say y_n who has arrived first. The algorithm is correct if Φ precludes the paths of other children but not the path of the fastest child. Let $<x, y_1, \ldots, y_n, w>$ be the path of the fastest child. Consider a path $<x, z_1, \ldots, z_m, w'>$ which arrives after y_n. Since the network is unambiguous, Φ either precludes $<x, y_1, \ldots, y_n, w>$ or $<x, z_1, \ldots, z_m, w'>$. That is, either

(1) there exists z_i $1 \leq i \leq m$ such that z_i is intermediary to $<y_n, w>$ in Φ, or

(2) there exists y_i $1 \leq i \leq n$ such that y_i is intermediary to $<z_m, w>$ in Φ.

However, since $<x, z_1, \ldots, z_m, w'>$ arrives after $<x, y_1, \ldots, y_n, w>$, Φ must preclude $<x, z_1, \ldots, z_m, w'>$.

4.8. Discussion

The shortest path algorithm can deal correctly with orthogonal class/property inheritance networks. As mentioned earlier, the class of orthogonal class/property inheritance networks is a subset of unambiguous networks. Therefore, the algorithm presented here can handle a larger class of networks than the simple shortest path algorithm. The run-time of our algorithm is proportional to the depth of the network while the run-time of the shortest path algorithm depends on the shortest path. So in a sense, the shortest path algorithm is more efficient. But in the worst case, both algorithms have the same efficiency.

The shortcoming of our algorithm is that it cannot detect ambiguity except in the special case where the ambiguous paths are of equal length. In contrast with the conditioning method, this algorithm does not need preprocessing. Conditioning is expensive. It has to compute the extension for the network and then add the appropriate links accordingly; and it has to detect if ambiguity is involved. When ambiguity is detected, it is assumed that there is an error and the conditioning process is aborted. As a result, when conditioning is successful, the resulting conditioned network has only one preferred extension. Thus, the shortest path algorithm is essentially used to reconstruct extension of unambiguous networks only. In addition, it has been remarked that conditioning is not amenable in the PMPM [Etherington 87b] while our algorithm can compute extension using the enhanced PMPM given an unambiguous network.

Comparing with Etherington's quasi-parallel algorithm, our algorithm is totally parallel. Another important difference is that our algorithm computes the preferred extension according to the inferential distance ordering whereas the quasi-parallel algorithm computes arbitrary extension(s). Computing extensions other than the preferred one may be useful because at times, we may want to analyze and compare different extensions.

The skeptical reasoner by Horty and his colleagues is based on a definition which is different from Touretzky's credulous definition. There are networks where the credulous reasoner would conclude ambiguity while the skeptical reasoner would conclude that we have an unambiguous situation. The difference results from a different conception of what preemption should be. We will discuss this in more detail in Chapter 5. For now, it is sufficient to know that when a credulous reasoner says there is no ambiguity, a skeptical reasoner will agree with it. In such a situation, our algorithm performs better since it is totally parallel while the skeptical reasoner is serial-parallel. Nevertheless, the skeptical reasoner is capable of detecting ambiguous situation while ours cannot, except in a special case. Also, both their algorithm and the one presented in this chapter employ a similar synchronization mechanism. The two algorithms differ in the way that conflict is handled. Let us consider unambiguous networks. When a node receives both an M_T and an M_F marker, an algorithm has to decide which marker to choose. In [Horty et al 87], a `check_preemption` algorithm is used. Essentially, it propagates markers through a subnet which is necessary to determine the relationship between the immediate child nodes, i.e., which preempts which. In addition, if a number of nodes can fire in one time step and there is conflict in the input, then the `check_preemption` algorithm has to be run for each of these nodes serially. In contrast, the algorithm presented here would solve the conflict by using the level numbers of the child nodes and hence no time is wasted for performing additional marker propagation. Also, nodes do not need to be processed serially since the decision is localized. The approach of enhancing the PMPM has the potential advantage of allowing a loosely coupled architecture without resorting to a central controller.

Chapter 5

Dealing with General Inheritance Networks

5.1. Introduction

In the previous chapter, we have seen how a slight extension to the PMPM can facilitate completely parallel processing for unambiguous networks. By restricting attention to only unambiguous networks, the solution is relatively straightforward. In this chapter, we will investigate on the requirements of a marker-propagation-style machine so that parallel algorithms can be written to deal with general networks.

Ideally, we want the behavior of our algorithm to be similar to the one that deals with unambiguous networks. Each node determines its own status by using only the local information provided by its child nodes. That is, we want a one-pass parallel algorithm that depends only on local information. We will discuss different alternatives in designing such an algorithm. Unfortunately, due to the interactions of multiple extensions of a general network, such an algorithm is difficult, if not impossible, to design.

In view of this difficulty, we follow the approach of [Horty et al 87]. We want to design a hybrid serial-parallel algorithm so that we can exploit the parallelism of the marker propagation as much as we can, and resort to serial processing only when it is necessary. The inheritance reasoner described in [Horty et al 87] is an off-path preemptor. We will discuss the difference between on-path and off-path preemption and why on-path preemption as described in [Touretzky 86] is more difficult to handle. We suggest an enhancement to the PMPM in which a set of link-checking and link-setting commands are provided so that a serial-parallel algorithm can be designed for general inheritance networks.

5.2. Requirement of the Algorithm

Before we discuss the algorithm for computing the preferred extension(s) for general networks, let us first examine the requirements of such an algorithm. As we have discussed earlier in this thesis, we want the marker-propagation machine to store real-world knowledge and to answer questions about the relationship among various entities.. When the user wants to know whether entity x is a subclass of entity y, he would pose the question as $subclass(x, y)$. If the information (i.e. the entities and their relationship to each other) stored in the machine forms an unambiguous network, then the answer to $subclass(x, y)$ will either be yes or no. The answer is yes if node y in the machine is marked with an M_T marker by the algorithm, otherwise the answer is no. However, when the underlying network is ambiguous, the machine has to decide which extension the user wants. This is because in one extension the answer to $subclass(x, y)$ may be yes but in another extension, the answer may be no. This would put a very heavy burden on the machine and the algorithm. Instead, we require our algorithm to return yes or no to $subclass(x, y)$ when the query-restricted network $\Gamma^{x,y}$ is unambiguous and to abort when $\Gamma^{x,y}$ is ambiguous. So the behavior of the marker propagation algorithm would be similar to TINA as mentioned in Section 3.3.2.

5.3. On-Path Verses Off-Path Preemption

Consider the status of a node y with respect to a node x in an inheritance network. We say the status of y is ambiguous if there are (preferred) extensions of the network where x is a y in one extension and x is not a y in another. In order to design marker-propagation algorithm, we need to know the kind of information that node y requires to determine its own status. The information required, however, depends on what inheritance system we have in mind. For example [Sandewall 86] and [Horty et al 87] promote a kind of preemption called off-path preemption while in [Touretzky 86], on-path preemption is used. The difference between these two types of preemption can best be illustrated by the network shown in Figure 5.1.

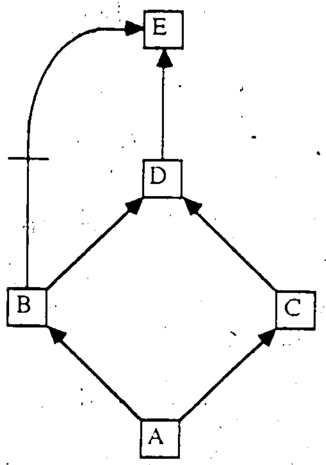


Figure 5.1 Network showing the difference between on-path and off-path preemption

In an inheritance system in which off-path preemption is performed, A is not an E. But for an on-path preemptor, whether A is an E or not is ambiguous. The argument for on-path preemption is as follows. There are three different paths leading from node A to node E: $A \rightarrow B \not\rightarrow E$, $A \rightarrow B \rightarrow D \rightarrow E$ and $A \rightarrow C \rightarrow D \rightarrow E$. The path $A \rightarrow B \not\rightarrow E$ overrides or preempts the path $A \rightarrow B \rightarrow D \rightarrow E$ since $B \not\rightarrow E$ is an immediate link and hence provides more specific information than the path $B \rightarrow D \rightarrow E$. However, $A \rightarrow B \not\rightarrow E$ does not preempt $A \rightarrow C \rightarrow D \rightarrow E$. The reason is that $B \not\rightarrow E$ does not lie on the path of $C \rightarrow D \rightarrow E$. Therefore we have two extensions, one in which A is an E and one in which A is not an E.

The argument for off-path preemption is that since $B \not\rightarrow E$ is a more specific piece of information, while $C \rightarrow D \rightarrow E$ is not, we should allow $A \rightarrow B \not\rightarrow E$ to override $A \rightarrow C \rightarrow D \rightarrow E$ also. Hence we can conclude that A is not an E. As we can see, in this type of preemption, a greater degree of preempting power is granted to immediate links. In doing so, we prefer a reasoner to draw definite conclusion rather that causing ambiguity.

Which of the two types of preemption is more intuitive is debatable. Their respective arguments seem to be equally sound. This represents one aspect of the "clashes of intuition" in the design of nonmonotonic multiple inheritance systems. For a more detailed discussion of these clashes, in particular off-path preemption verses on-path preemption, the reader is referred to [Touretzky et al 87].

For off-path preemption, the amount of information that a node requires to determine its status is more localized. For example, consider node E in Figure 5.1. There are two immediate links to it, namely $B \not\rightarrow E$ and $D \rightarrow E$. In off-path preemption, in order to check whether $B \not\rightarrow E$ overrides $D \rightarrow E$, all we need to do is to check if there is any path from node B to node E that passes through node D. If there is, $B \not\rightarrow E$ overrides $D \rightarrow E$. In other words, we only need to know the interrelationship between the immediate links to a node in order to determine the status. We do not have to worry about links such as $C \rightarrow D$. For on-path preemption, however, not only do we need to consider the relationship between the immediate links, we also need to consider how the immediate links are actually constructed. For the network in Figure 5.1, the immediate link $D \rightarrow E$ of node E can be constructed either as $A \rightarrow B \rightarrow D \rightarrow E$ or $A \rightarrow C \rightarrow D \rightarrow E$. According to on-path preemption, we need to consider whether or not $B \not\rightarrow E$ preempts $A \rightarrow B \rightarrow D \rightarrow E$ and $A \rightarrow C \rightarrow D \rightarrow E$ separately. This makes the task of designing on-path preemptor much more difficult.

5.4. Alternatives in Designing Algorithms for General Inheritance Networks

Suppose we want to design a one-pass parallel algorithm so that each node can determine its status by using only the information carried by its immediate child nodes. A naive way is to let each node pass the whole trace of the paths that lead to it [Fahlman et al 81]. For instance, node D in Figure 5.1 would pass something like $[(+A,+B,+D),(+A,+C,+D)]$ to node E and node B would pass $[(+A,+B)]$ to node E. Node E in turn has to determine which input should override which one from these information structures. As we go up the network, the amount of information grows. This is because as we go up the network, the number of paths that reach a particular node increases and we have to record the construction of each of these paths separately. Since each node is a processing element, adopting this naive approach would mean that each processing element is a complicated computer capable of analyzing complex information. This excessive requirement on the processing elements makes this simple and naive approach practically impossible.

The next attractive approach is similar to the one taken in designing the algorithm in Chapter 4: pass around some additional information that does not grow so that each node can use this additional information to make the correct inference. Since we do not want the amount of information to grow as we go up the network, this approach amounts to encoding the actual construction of each path in to a nice, small piece of information. This piece of information may be in the form of an additional number or an additional marker. What such an encoding scheme wants to achieve is to transform some unstructured information which is growing rapidly as the derivation of a path gets longer, to some information which is constant and structured. To make this clear, let us consider node D in Figure 5.1 again. The encoding scheme has to condense $[(+A, +B, +D), (+A, +C, +D)]$ into some structured information and let node D pass this to node E. Node E then has to extract from that information to do the inference. Such an encoding scheme seems impossible to design. Even if it is possible, encoding and decoding information would be complex, which again require complicated processing elements.

From the above discussion, we can see that although one-pass parallel algorithm for inheritance reasoning is attractive for its tremendous efficiency, it is practically impossible to design. The common characteristic of the above approaches is to let each node pass sufficient information to its parent nodes so that the parent nodes can make further inference. The problem is that the amount of information grows too rapidly as we go up the hierarchy. To get around this problem, we have to generate the information "on-line". This is the approach used in the skeptical reasoner in [Horty et al 87]. The idea is that when a node is unsure of its status, a check-preemption algorithm is run. Basically, the algorithm performs some marker propagation through part of the network to extract the necessary information to determine the status. Since the skeptical reasoner performs off-path preemption, the check-preemption algorithm is relatively simple. It only needs to determine if one immediate link (to the node in question) overrides other immediate links or not. This serial-parallel approach is obviously less efficient than a one-pass parallel algorithm because when a node has trouble in determining its status, other nodes have to wait until it has solved the problem. However, in view of the difficulty for designing one-pass parallel algorithm, it seems that we have to settle for such a serial-parallel approach. In here, we want to design an algorithm which performs on-path preemption according to the definition in [Touretzky 86] rather than off-path preemption.

5.5. Enhancing the PMPM

In this section, we discuss the enhancement of the PMPM that is required for the parallel inheritance algorithm described in the next section.

As mentioned in the previous section, when problem arises, an off-path preemptor, such as the skeptical reasoner presented in [Horty et al 87], performs some marker propagation through part of the network to extract the necessary information to solve the problem. However, for an on-path preemptor, such as the credulous reasoner presented in [Touretzky 86], the simple check-preemption algorithm is not sufficient. The reason is that in order to extract the necessary information, we may need to check the entire network. To get around the problem, we suggest a simple enhancement of the PMPM which allows us to deposit information onto the links as we go up the network.

In the PMPM, the links between the nodes are either of type "$\rightarrow$" or "$\not\rightarrow$". While we can mark a node with a node marker, we cannot explicitly mark a link with a link marker. In our enhancement, commands are provided so that we can mark a link with a specific link marker, test whether a link marker is on a link etc. That is, we provide a set of link-marking and link-checking commands which are counterparts of those node-marking and node-checking commands. This enhancement allows us to indicate whether a link is redundant, or preempted, etc. Using these information, more efficient inheritance algorithms can be designed.

The commands that we can use after the enhancement are:

set_link[$L_1, \ldots, L_n$]

clear_link[$L_1, \ldots, L_n$]

any_on_link[$L_1, \ldots, L_n$]

on_link[$L_1, \ldots, L_n$]

any_off_link[$L_1, \ldots, L_n$]

off_link[$L_1, \ldots, L_n$]

where L_i's are link markers.

5.6. A Serial-Parallel Algorithm for General Inheritance Networks

In this section, we present a serial-parallel algorithm for general nonmonotonic multiple inheritance networks. To facilitate understanding of the algorithm, we first discuss the intuition behind the design of the algorithm.

We call the subnet formed by the immediate links of a given node w the immediate subnet of w. An immediate subnet of a node w is the net formed by the immediate-child nodes $u_1, \ldots, u_n$ of w and all the nodes which lie in the paths between u_1 and w, u_2 and w, . . . , u_n and w. For instance, consider node G in Figure 5.2. Node B and node F are immediate child nodes of G. The immediate subnet of node G is marked by dotted arrows. Notice that node E is also in the immediate subnet since there is a path between node B and node G through node E. When a node wants to determine its status, not only do we need to consider the immediate subnet, but we also need to consider all the paths leading to each node in the subnet. For instance, in Figure 5.2, we can easily determine that $B \not\rightarrow G$ preempts $B \rightarrow E \rightarrow F \rightarrow G$ from the immediate subnet of G.

Besides, we need to consider alternate paths from node A to node G, which do not go through node B, such as $A \to C \to E \to F \to G$ and $A \to D \to F \to G$. The idea of the algorithm is to do the checking in two steps. First we check if there is any immediate link that is not preempted by other immediate links. If there is more than one such link and that their input conflicts with each other, then the node in question is ambiguous. On the other hand, if there is only one such link $u \to w$, then we have to carry out the second step. Essentially, the purpose of the second step is to find out if there is any other alternate path that does not go through $u \to w$. To do this, we propagate a special marker from those links whose head node is in the immediate subnet of w, except those that are part of the immediate subnet of w and those that lead to node u. If such a marker reaches the node in question, then we know that there is a path that the link $u \to w$ cannot preempt. If the input from the path does not agree with that of $u \to w$, then the status of w is ambiguous.

Consider [illegible] again. After we decided that the link $B \not\to G$ preempts $B \to E \to F \to G$, we start propagating a special marker from links $C \to E$ and $D \to F$ up to G. We do not start propagating the marker from link $A \to B$ since B is the tail node of the only non-preempted immediate link of G.

A number of subtleties need to be considered. In the second step described above, the condition for which a link would propagate the special marker is not really sufficient. For instance, consider the network in Figure 5.3. According to the definition in [Touretzky 86], node A is not a node D. Notice that link $A \to C$ is not in the immediate subnet of node D. But if we allow the link $A \to C$ to propagate the special marker up to node D, we would conclude that the status of D is ambiguous. Notice also that the link $A \to C$ is a redundant link. Therefore, we want to restrain redundant links, such as $A \to C$, from propagating the special marker. This is where the link-setting commands are used. Redundant links are marked by some special link markers as we go up the network.

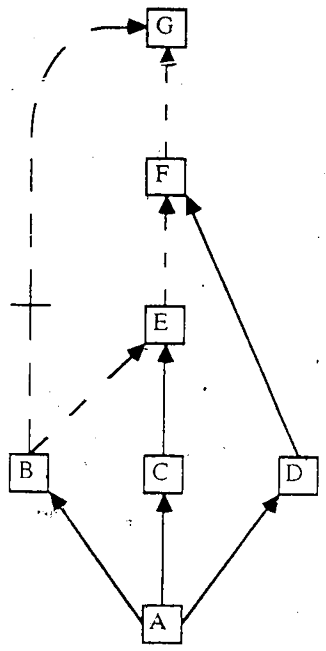


Figure 5.2 Paths have to be considered separately for on-path preemption

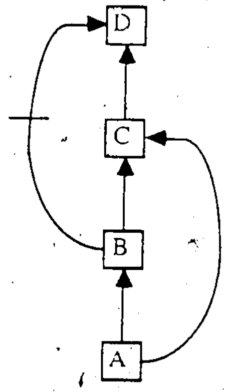


Figure 5.3 Suppressing redundant link

There are other links which we also do not want them to propagate the special marker, such as those that have been preempted by others and those that do not form a well-formed path. For example, in the network shown in Figure 5.4, we do not want link $C \rightarrow D$ to pass the special marker upward since we cannot go from node A to node F through node C. The path $A \not\rightarrow C \rightarrow D \rightarrow E \rightarrow F$ is not well-formed. These special links are also marked as we go up the network.

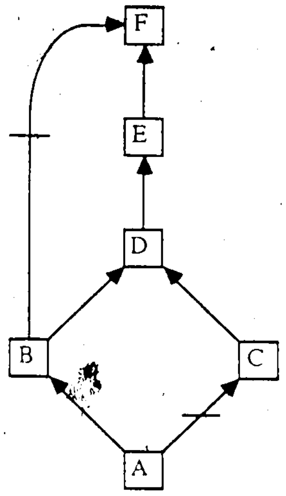


Figure 5.4 Suppressing fake path

5.6.1. Summary of the Markers Used

In the algorithm, a total of seventeen node markers and three link markers are used. In this subsection, we describe the purposes of these markers in order to ease the understanding of the algorithm. The markers are presented roughly according to the order in which they first appear in the algorithm.

M_x: To identify node x given a call to `subclass(x,y)`, i.e. node x is marked by the marker M_x.

M_y: To identify node y given a call to `subclass(x,y)`, i.e. node y is marked by the marker M_y.

M_2: Used in the procedure `trim_for_query(x,y)` to mark the nodes in the query-restricted network $\Gamma^{x,y}$ (see Section 3.5).

M_{Fire}: Used in the procedure `select_node()` to mark all the nodes whose inputs have all arrived (see Section 4.4).

M_{old}: Used in the procedure `select_node()` to mark all the nodes that have been fired.

M_M: Used to mark the nodes which are marked by M_{Fire} and have more than one child node.

M_c: When there is more than one node which is marked by M_{Fire} and has more than one child node, we need to process these nodes separately (Section 5.8 will discuss this in further detail). The M_c marker is used to identify the node which the algorithm is currently processing.

M_{dir}: Used to mark those nodes which are the immediate child nodes of a given node with the marker M_c.

M_{pre}: Used to identify those nodes that are preempted.

Consider the network in Figure 5.5.

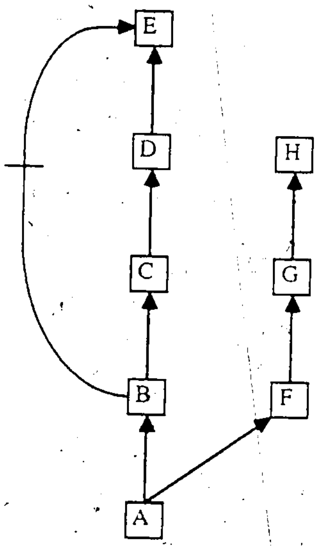


Figure 5.5 Network showing how some of the markers are used

Suppose that a call to `subclass(A,E)` is made, i.e., we want to know if A is a subclass of E. Hence, node A and node E are marked with M_x and M_y respectively. Notice that nodes F, G and H are not relevant in determining the relationship between A and E. Therefore the `trim_for_query(A,E)` procedure will only mark nodes A, B, C, D and E with the marker M_2. Now suppose that we are currently processing node E. Nodes A, B, C and D are marked with M_{old} since they have been processed already and node E is marked with M_{Fire} by `select_node()`. Node E is also marked with M_M since it has more that one immediate child node, namely B and D. Finally, node E is marked with M_c since we are currently processing it. Nodes B and D are marked with M_{dir} since they are the immediate child nodes of E. After some processing, nodes C and D are marked with M_{pre} which indicates that they are preempted.

M_{dorm}: This is used for checking if there is a redundant link. If a node w has more than one immediate child node and if any one of the child nodes is preempted, then node w is marked with M_{dorm}.

L_{red}: To mark a link which is redundant (see Section 2.6 for the definition of a redundant link).

M_{tt}: For a given node w with an M_c marker, it is marked with M_{tt} if there is a positive link between itself and any one of its immediate child nodes which is not marked with a M_{pre} marker.

M_{ff}: For a given node w with an M_c marker, it is marked with M_{ff} if there is a negative link between itself and any one of its immediate child nodes which is not marked with a M_{pre} marker.

Using the M_{tt} and M_{ff} markers, some cases of ambiguity such as the one shown in Figure 5.6 can be detected.

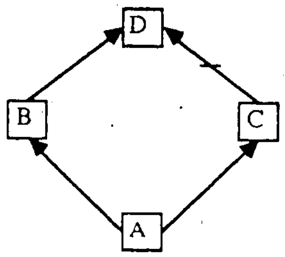


Figure 5.6 Using M_{ti} and M_{ff} to determine ambiguity

Suppose that we are processing node D. Both nodes B and C are marked with M_{dir} but not with M_{pre}. Hence node D is marked with both M_{tt} and M_{ff}. This indicates that whether node A is a subclass of node D is ambiguous because neither B nor C is preempted.

M_{ab1}: Suppose that we are processing a node w with multiple child nodes and that only one of these child nodes, u, is not preempted. The M_{ab1} marker is used to check if there is any other path from node x to node w which does not go through node u. Basically, the nodes which are immediate child nodes of the nodes in the immediate subnet of w is marked with M_{ab1}, except

1. those nodes which are part of the immediate subnet of w
2. those that have direct links to node u.
3. those that are the head nodes of redundant links.
4. those that are the head nodes of preempted links.
5. those that do not form a well-formed path with node x.

The algorithm then tries to propagate M_{ab1} up to node w. If w receives the marker, then there is a positive path from x to w which does not go through node u.

M_{ab2}: Used in conjunction with M_{ab1} to handle negative links. If the tail of a negative link is marked with an M_{ab1} marker, an M_{ab2} marker is propagated to the head of the link. If w receives the marker, then there is a negative path from x to w which does not go through node u.

The reason why we need both M_{ab1} and M_{ab2} markers is that we do not want either of these markers to reach node w when there is no actual or well-formed path between node x and w. Consider Figure 5.7.

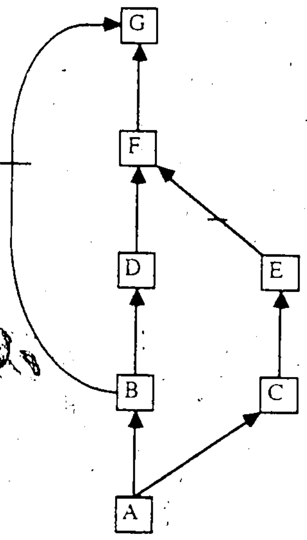


Figure 5.7 Network showing the use of the M_{ab1} and M_{ab2} markers

Suppose that we are processing node G. Node E is marked with M_{ab1} since it is an immediate child node of F which is in the immediate subnet of node G. However, there is no path between node A and G through E. ($A \rightarrow C \rightarrow E \not\rightarrow F \rightarrow G$ is not a well-formed path. See Section 2.6.)

L_{pre}: Given a node w, the link $u \rightarrow w$ is marked as L_{pre} if u is marked with M_{pre}. Notice that u is marked with M_{dir} too since it is an immediate child node of w. Therefore, L_{pre} is used to mark an immediate link as being preempted.

M_T: A node w is marked with an M_T marker if it is a superclass of x.

M_F: A node w is marked with an M_F marker if it is not a superclass of x.

M_D: When the path between node w and node x is not well formed, w is marked with M_D. Node w is not a superclass of node x.

L_{nopath}: A link $u \rightarrow w$ ($u \not\rightarrow w$) is marked with a L_{nopath} marker if the path $x \rightarrow \cdots \rightarrow u \rightarrow w$ ($x \rightarrow \cdots \rightarrow u \not\rightarrow w$) is not well-formed.

5.6.2. Structure of the Algorithm

Given a call to the procedure `subclass(x,y)`, it first marks the initial node x with markers M_T. It then calls the procedure `trim_for_query(x,y)` (see Section 3.5) so that all the relevant nodes are marked with marker M_2.

The main loop from line 5 to line 46 is repeated until either node y is correctly marked or ambiguity is detected as markers are propagated from node x to node y. The main loop is composed of two parts. The `for` loop between line 9 and line 41 is used for two purposes: to detect ambiguous situation and to mark special links with some special markers. The second part of the main loop, from line 42 to line 45, is used for propagating markers upward.

The `for` loop can also be decomposed into two subparts. The commands between line 10 and line 23 are used to check if there is only one link that is not preempted. If there is more than one and that their inputs conflict with each other, then the network is ambiguous and the procedure is aborted at line 23. The commands between line 25 and line 40 are used to check if there is any other path from node x to the current node that does not go through the only non-preempted link. If there is, and that the input from the non-preempted link conflicts with the input of the path, the network is ambiguous and the procedure is stopped.

```
Procedure subclass(x,y: node)
Begin
  name[x] $\Rightarrow$ set[$M_T$];
  trim_for_query(x,y);
  loop
    clear[$M_{dir}$, $M_{pre}$, $M_{tt}$, $M_{ff}$, $M_c$];
    select_node();
    on[$M_{Fire}$], more_than_one_child() $\Rightarrow$ set[$M_M$];
    for $c$ in <on[$M_M$]> do begin
      set[$M_c$];
      link_type[$\rightarrow$, $\not\rightarrow$], on_tail[$M_T$], on_head[$M_c$] $\Rightarrow$ set_tail[$M_{dir}$];
      loop
        link_type[$\rightarrow$], any_on_tail[$M_{dir}$, $M_{pre}$], on_head[$M_T$]
           any_on_head[$M_{old}$, $M_{Fire}$] $\Rightarrow$ set_head[$M_{pre}$];
      endloop

      link_type[$\rightarrow$, $\not\rightarrow$], on_tail[$M_{pre}$], on_head[$M_c$] $\Rightarrow$ set_head[$M_{dorm}$];
      link_type[$\rightarrow$, $\not\rightarrow$], on_head[$M_T$], off_head[$M_F$, $M_D$], on_tail[$M_{dir}$]
           on_head[$M_{dorm}$], off_tail[$M_{pre}$] $\Rightarrow$ set_link[$L_{red}$];

      link_type[$\rightarrow$], on_tail[$M_{dir}$], off_tail[$M_{pre}$] $\Rightarrow$ set_head[$M_{tt}$];
      link_type[$\not\rightarrow$], on_tail[$M_{dir}$], off_tail[$M_{pre}$] $\Rightarrow$ set_head[$M_{ff}$];
      on[$M_c$], on[$M_{tt}$], on[$M_{ff}$] $\Rightarrow$ abort();

      link_type[$\rightarrow$], on_head[$M_{pre}$], on_tail[$M_2$], off_tail[$M_{pre}$, $M_{dir}$],
           off_link[$L_{red}$, $L_{nopath}$, $L_{pre}$] $\Rightarrow$ set_tail[$M_{ab1}$];
      loop
        link_type[$\rightarrow$], on_tail[$M_{ab1}$], any_on_head[$M_{pre}$, $M_c$],
        [illegible], off_link[$L_{red}$, $L_{pre}$] $\Rightarrow$ set_head[$M_{ab1}$];
        link_type[$\not\rightarrow$], on_tail[$M_{ab1}$], any_on_head[$M_{pre}$, $M_c$],
            off_link[$L_{red}$, $L_{pre}$] $\Rightarrow$ set_head[$M_{ab2}$];
      endloop;

      on[$M_c$, $M_{tt}$, $M_{ab2}$], $\Rightarrow$ abort();
      on[$M_c$, $M_{ff}$, $M_{ab1}$], $\Rightarrow$ abort();

      on[$M_c$] $\Rightarrow$ clear[$M_T$, $M_F$];
      on[$M_c$], on[$M_{tt}$], off[$M_{ff}$] $\Rightarrow$ set[$M_T$];
      on[$M_c$], on[$M_{ff}$], off[$M_{ff}$] $\Rightarrow$ set[$M_F$];
      link_type[$\rightarrow$, $\not\rightarrow$], on_tail[$M_{dir}$, $M_{pre}$] $\Rightarrow$ set_link[$L_{pre}$];
    end for}
    link_type[$\rightarrow$], on_tail[$M_{Fire}$, $M_T$] $\Rightarrow$ set_head[$M_T$];
    link_type[$\not\rightarrow$], on_tail[$M_{Fire}$, $M_T$] $\Rightarrow$ set_head[$M_F$];
    link_type[$\rightarrow$, $\not\rightarrow$], on_tail[$M_{Fire}$], any_on_tail[$M_F$, $M_D$]
      $\Rightarrow$ set_head[$M_D$], set_link[$L_{nopath}$];
  endloop {main loop}
End.
```

Figure 5.8 A parallel algorithm for general inheritance networks

5.7. Examples

5.7.1. Example 1

The problem that we want to solve here is the familiar network shown in Figure 5.9.

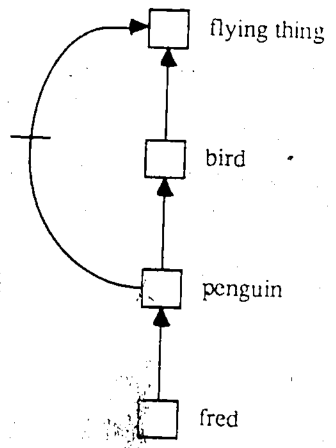


Figure 5.9 The problem for the first example

It is an unambiguous network. A redundant link (*fred* → *bird*) is present but its presence does not change the fact that *fred* is not a *flying thing*. Since this is an unambiguous network, the algorithm shown in Chapter 4 can also produce the correct answer.

The algorithm starts by marking node *fred* with M_T (Figure 5.10a). Since it is a leaf node, it has no child. Therefore, it is not marked with M_M at line 8. The `for` loop between line 9 and line 41 is skipped. An M_T marker is then propagated to nodes ***penguin*** and ***bird*** at line 42. We then enter the main loop again. The `select_node()` procedure marks node ***penguin*** with M_{Fire} but not node *bird* since the input from node *penguin* to *bird* has not arrived yet. Again, since node *penguin* has only one child node, it is not marked with M_M. The `for` loop is skipped. An M_T marker is propagated to node *bird* and an M_F marker is propagated to node *fly thing* at line 42 and 43 respectively (Figure 5.10b). At the next iteration, node *bird* is the only node that is marked with M_{Fire} by `select_node()`. In addition, it is marked with M_M at line 8 because it has two child nodes. Th[illegible] time we enter the `for` loop.

We first mark node *bird* with M_c. Line 11 marks both nodes *penguin* and *fred* with M_{dir}. The result of the loop in line 12 to line 15 is that node *penguin* is marked with M_{pre}. After executing the commands between line 17 and line 19, link *fred* $\rightarrow$ *bird* is marked with L_{red}. An M_{tt} marker is then propagated to node *bird* at line 21 (Figure 5.10c). Since there is no alternate path from node *fred* to *bird*, nothing happens between line 25 and line 35. Node *bird* is marked as M_T at line 38 and link *penguin* $\rightarrow$ *bird* is marked as L_{pre} at line 40 (Figure 5.10d). Node *penguin* is the only node that has M_M marker on, therefore we exit the `for` loop and a M_T marker is passed to node *fly thing* at line 42.

We now enter the final iteration. Node *fly thing* has more than one child node and so the commands inside the `for` statement are executed. As a result of executing line 10 to line 15, node *bird* is marked both as M_{dir} and M_{pre} and node *penguin* is marked with M_{dir}. Node *fly thing* is marked with M_{ff} at line 22 but not with M_{tt} at line 21 (Figure 5.10e). Since link *fred* $\rightarrow$ *bird* is marked with L_{red} previously, lines 25 to 26 are not executed. Hence, *fred* is not marked with M_{ab1}. Nothing happens from line 27 to line 35, either; because no node has been marked with M_{ab1}. Node *fly thing* is marked with M_F at line 39 and line *bird* $\rightarrow$ *fly thing* is marked with L_{pre} at line 40 (Figure 5.10f). No further commands are executed and the process is done.

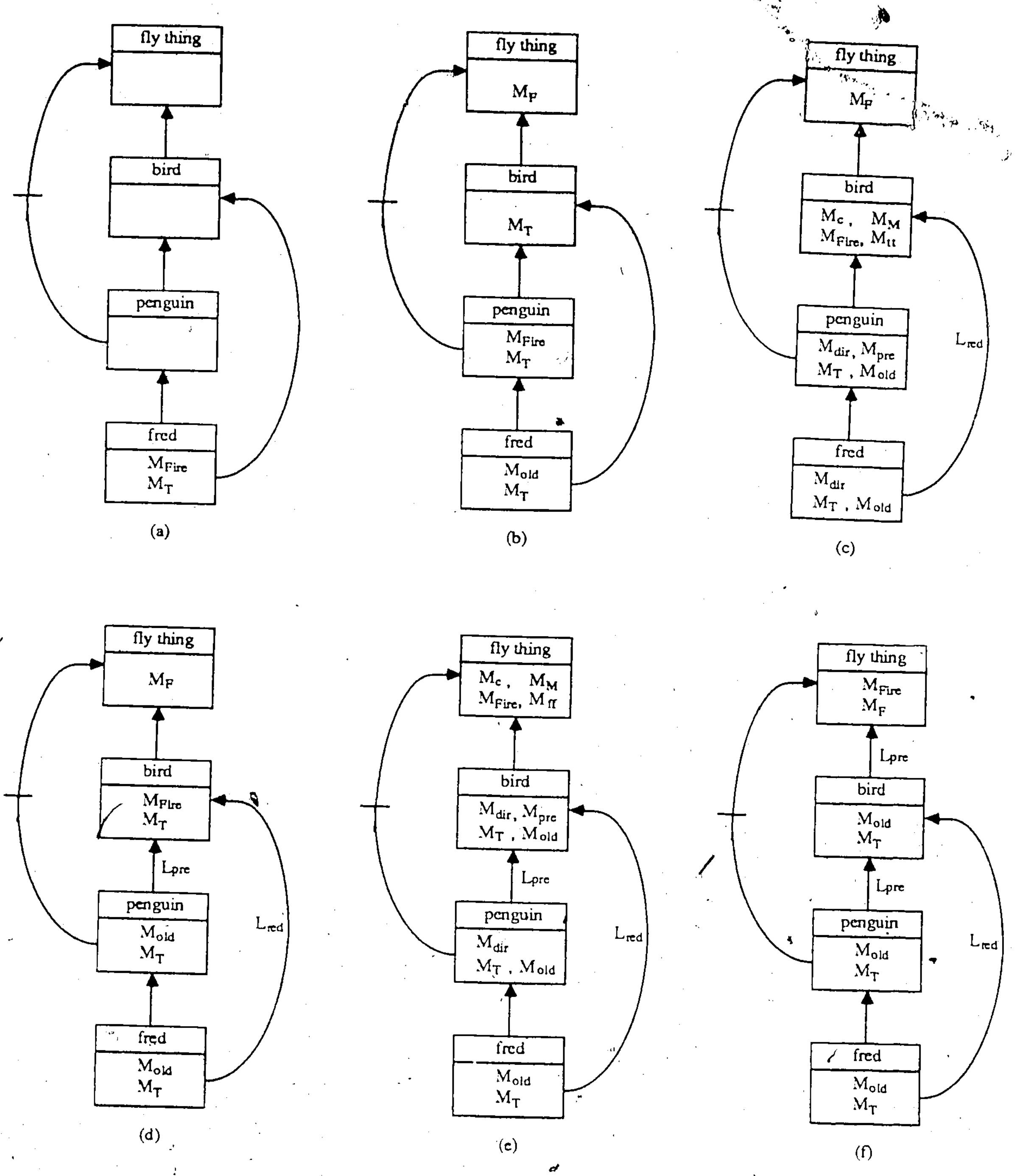


Figure 5.10 Example 1

5.7.2. Example 2

The problem that we want to solve in the second example is shown in Figure 5.11. It is originally presented in [Touretzky et al 87]. It is used as a "real" example which demonstrates the usefulness of on-path preemption. Using off-path preemption, *Tom* is not a *Beer Drinker*. But since whether *Tom* drinks beer or not really depends on the rate of beer-drinking among *Marines* and that of a *Chaplain*, Touretzky and his colleagues argue that we should be more cautious and conclude that we have an ambiguous situation, just like what an on-path preemptor would conclude.

Since this is an ambiguous network, the algorithm in Chapter 4 behaves like a shortest path algorithm and concludes that *Tom* is not a *Beer Drinker*.

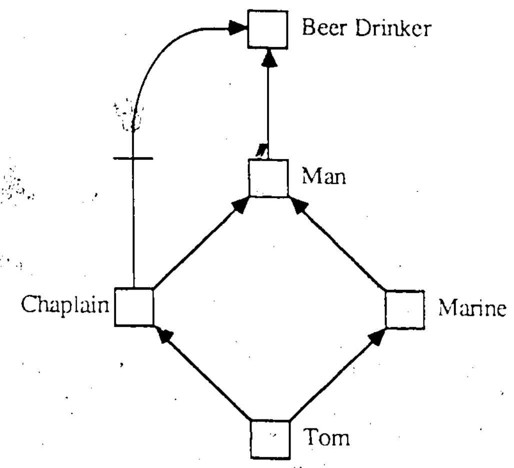


Figure 5.11 The problem for the second example

Since nodes *Tom*, *Chaplain* and *Marine* all have only one child node, the first two iterations are simple. The result is that nodes *Tom*, *Chaplain* and *Marine* are marked with M_T and node *Beer Drinker* is temporarily marked with M_F (Figure 5.7a). In the next iteration, node *Man* is marked with M_{Fire} and M_M and we enter the `for` loop. As a result of executing line 10 to line 15, both node *Chaplain* and *Marine* are marked with M_{dir}. After line 21, node *Man* is marked with M_{tt} (Figure 5.7b). Node *Man* is then marked as M_T at line 38 (Figure 5.7c).

In the final iteration, only node *Beer Drinker* is marked with M_{first}. After executing line 10 to line 15, node *Chaplain* is marked as M_{dir} and node *Man* is marked with M_{dir} and M_{pre}. Node *Beer Drinker* is marked with M_{ff} at line 22 (Figure 5.7d). The link *Marine* $\rightarrow$ *Man* satisfies the conditions at line 25 and 26 and therefore node *Marine* is marked with M_{ab1} (Figure 5.7e). After the loop from line 27 to line 31 is executed, node *Beer Drinker* is marked with M_{ab1} (Figure 5.7f). This indicates that there is an alternate path from node *Tom* to *Beer Drinker*, namely *Tom* $\rightarrow$ *Marine* $\rightarrow$ *Man* $\rightarrow$ *Beer Drinker*, which does not go through node *Chaplain*. Since the inputs from *Chaplain* $\not\rightarrow$ *Beer Drinker* and the alternate path are in conflict, we have an ambiguous situation. The procedure is aborted at line 35.

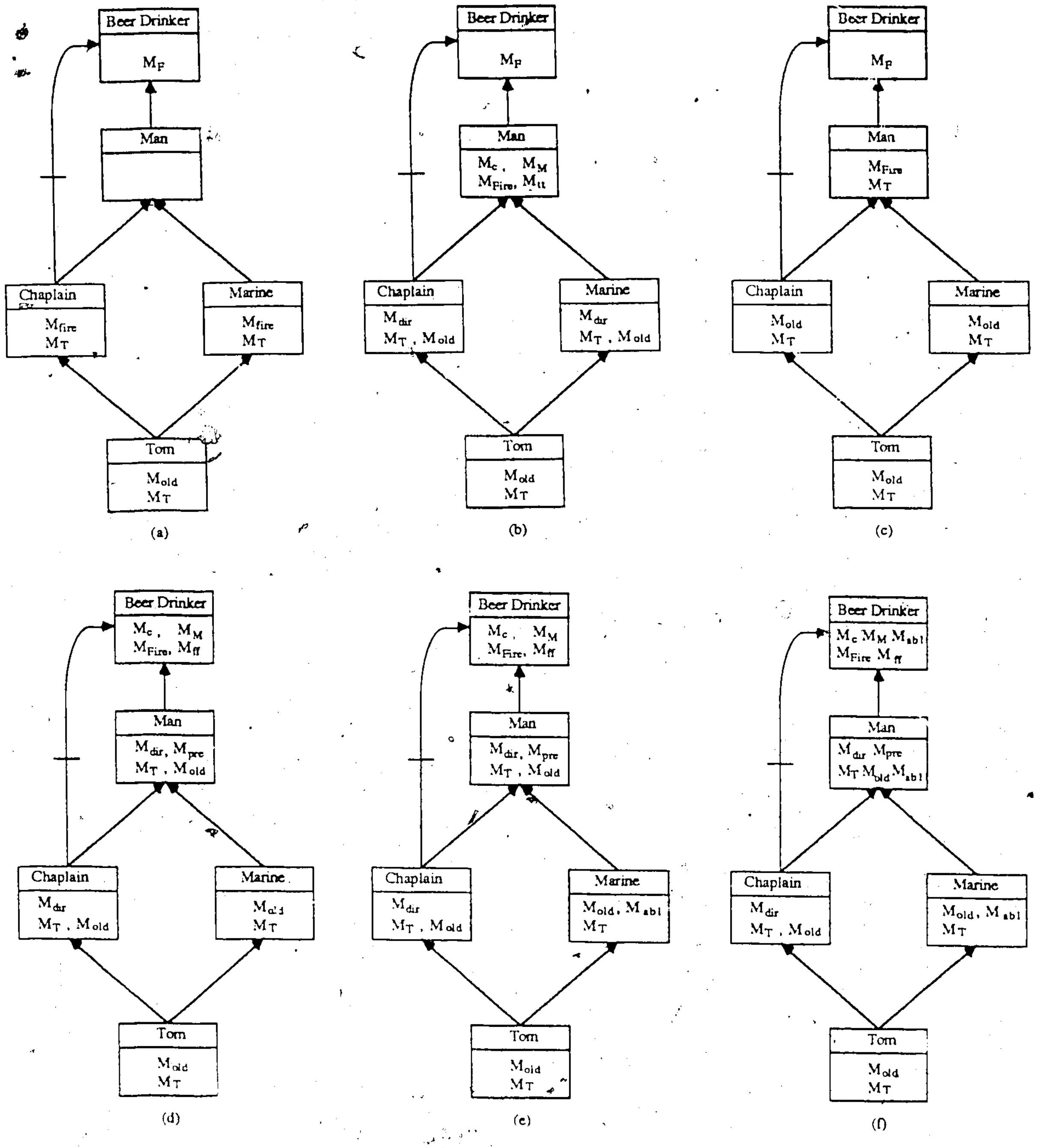


Figure 5.12 Example 2

5.8. The Serial Process in the Algorithm

At any moment, when there is more than one node which has more than one child node and that all of the inputs have arrived, we have to process these nodes one by one serially. That is the purpose of the `for` loop in the procedure. To illustrate why this serial processing is required, consider the network shown in Figure 5.13.

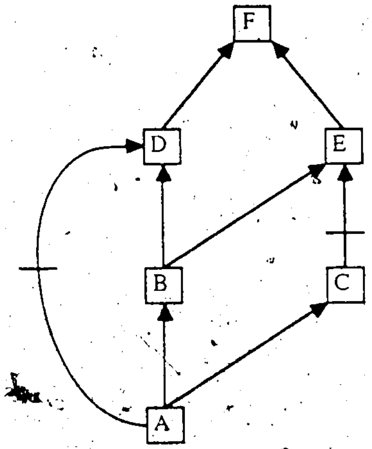


Figure 5.13 Network showing the serial process of the algorithm

Notice that node D should be marked with M_F while node E is ambiguous. After the first iteration, nodes A, B and C are marked with M_T. In the next iteration, node D and E are marked with M_{Fire} by `select_node()`. Suppose we process the two nodes at the same time. Nodes A, B and C are marked with M_{dir}. The result of executing the loop between line 12 and 15 is that node B is marked with M_{pre}. This will prohibit node B from propagating a M_{tt} marker to node E. Node E will then be marked incorrectly as M_F since there is no conflicting input. Processing nodes D and E one after the other prevents such a problem.

5.9. Correctness

In this section, we provide an intuitive argument as to why the algorithm works according to the inheritance reasoning as defined in [Touretzky 86].

First of all, we want to argue that only redundant links are marked with the L_{red} marker from line 17 to 19. Recall that a link $<x, y_n>$ is redundant if the path $<x, y_1, \ldots, y_{n-1}, y_n>$ is inheritable in the given network. Notice that for a node y_n to have a redundant link, there must be a node y_{n-1} where the link $<y_{n-1}, y_n>$ is in the network and that y_{n-1} is marked with M_{pre} by line 10 through 15. That is, there must be a path from x to y_n through y_{n-1} in the network. Line 17 makes sure that this condition is satisfied. If the condition is satisfied, node y_n is marked with M_{dorm}. Another condition is that there should only be positive links leading to y_n. Obviously, if there are different types of link leading to y_n, there is no redundant link. In addition, only immediate links to node y_n can be redundant. Line 18 and 19 make sure that these conditions are satisfied. If they are, we have found a redundant link and it is marked with L_{red}.

Next, we want to argue that given a link $<y_{n-1}, y_n>$, it is marked with L_{nopath} if the path $<x, y_1, \ldots, y_{n-1}, y_n>$ is not well-formed. A path $<x, y_1, \ldots, y_{n-1}, y_n>$ is not well-formed if any of the links $<x, y_1>$, $<y_i, y_{i+1}>$ where $1 \le i < n$, is a negative link. In line 44 and 45, whenever a node y_i is marked with an M_F (i.e. $<y_{i-1}, y_i>$ is a negative link), an M_D marker is propagated to y_{i+1}, and in turn to y_{i+2}, etc. The link $<y_i, y_{i+1}>$ is also marked as L_{nopath}. The M_D and L_{nopath} markers are propagated until node y_n is marked with M_D and the link $<y_{n-1}, y_n>$ is marked with L_{nopath}.

Finally, we want to argue that links $<w_1, y_n>, \ldots, <w_m, y_n>$ are marked with L_{pre} markers if nodes $w_1, \ldots, w_m$ are preempted by a link $<x, y_n>$. At line 11, nodes $x, w_1, \ldots, w_m$ of the direct links $<x, y_n>, <w_1, y_n>, \ldots, <w_m, y_n>$ to y_n are marked with M_{dir}. That is, the tail nodes of the direct links of y_n are marked with M_{dir}. Suppose that link $<x, y_n>$ preempts all other links. The loop from line 12 to 15 marks all the nodes with M_{pre} which are intermediary to $<x, y_n>$, except node x itself. All links to y_n, whose tail node is marked with an M_{pre} marker, is marked with a link marker L_{pre} at line 40.

In determining the status of a node y_n, we need to consider all the paths from node x to y_n that are well-formed, non-redundant and non-preempted. Lines 10 through 15 and lines 21 through 23 in the algorithm consider a subset of these paths. They determine if there is an immediate link $<y_{n-1}, y_n>$ which preempts other immediate links. Obviously, if there is no such a link and that the inputs from the immediate links conflict with each other, then the status of y_n is ambiguous. But if there is indeed an immediate link $<y_{n-1}, y_n>$ that preempts others, we need to consider the remaining subset of the paths which are well-formed, non-redundant and non-preempted. This subset may or may not be empty. The paths in the subset are found by propagating the M_{ab1} and M_{ab2} markers to node y_n. If M_{ab1} reaches node y_n, that means that there is a positive path from x to y_n which does not go through y_{n-1}. If the link $<y_{n-1}, y_n>$ is negative, then the status of y_n is ambiguous. This condition is checked by line 34. If M_{ab2} reaches node y_n, that means that there is a negative path from x to y_n which does not go through y_{n-1}. If the link $<y_{n-1}, y_n>$ is positive, then the status of y_n is ambigious. This condition is checked by line 35.

5.10. Discussion

In this chapter, we have discussed and analyzed certain problems concerning the design of parallel inference algorithms for inheritance reasoning. A simple modification to the traditional parallel marker propagation machine is suggested and an algorithm running on such a machine is presented. In this section, we compare this algorithm with other related parallel algorithms for inheritance reasoning.

Due to the simplicity of the PMPM, the shortest-path algorithm is the most natural algorithm to be implemented on such a machine. However, the shortest-path algorithm does not always produce the correct answer for nonmonotonic multiple inheritance systems. The best it can do is to deal with orthogonal class/property inheritance networks. Our algorithm is certainly less efficient than the simple *upscan* algorithm. But we sacrifice speed in order that we can deal with general inheritance networks. A network has to be "conditioned" in order that the *upscan* algorithm can run correctly. The conditioning is done by a reasoner called TINA. It has another function: to compute the preferred extension(s) of a given inheritance network. If multiple extensions are detected, TINA will respond with an error message. If there is only one extension, TINA will modify the network so that the *upscan* algorithm can work correctly. Hence the PMPM and the *upscan* algorithm are used only to reconstruct the extension while the difficult job of computing the extension is left for TINA which does not do any parallel processing. The parallel algorithm presented in this chapter does not require the given inheritance network to be modified. The extension is computed, rather than reconstructed, by running the algorithm on the enhanced PMPM. No expensive reconditioning is required when the given inheritance network is updated.

The skeptical approach [Horty et al 87] to inheritance reasoning will always produce a unique extension for any inheritance network. This relieves the burden of a skeptical reasoner to worry about multiple extensions. In addition, as we have discussed earlier in this section, the off-path preemption performed by the skeptical reasoner is easier to deal with by marker-propagation than on-path preemption. The credulous reasoner presented in [Touretzky 86] has caused a lot of problems for the PMPM since it has to consider multiple extensions and it is an on-path preemptor. We have shown how a slight extension of the PMPM allows a serial-parallel algorithm for the inheritance reasoner defined in [Touretzky 86].

Chapter 6

Summary and Future Research

6.1. Summary

Organizing information in the form of a hierarchy is a common practice in knowledge representation. The most important concept in such representation systems is inheritance: a subclass inherits all the properties of its superclass. The concept of inheritance is extremely useful as it allows us to represent knowledge implicitly. The properties of a superclass are not repeated in a subclass. Rather, when the information is needed, those properties are extracted from the superclass. Economy in storage is achieved.

In a monotonic multiple inheritance system or a tree-structured inheritance system, the problem of whether one class is a subclass of another does not arise. Another advantage of these systems is that parallel processing can be achieved by using simple parallel architectures such as the PMPM. Nevertheless, these systems are too restrictive.

Nonmonotonic multiple inheritance systems are the most general inheritance systems. They allow a class to inherit properties from multiple superclasses and exceptions to those inherited properties are also permitted. These flexibility makes them very attractive for knowledge representation. However, these flexibility does not come without a cost. Firstly, given an inheritance network, it is not as clear as other simpler systems whether a class is a subclass of another. Different system designers may have different intuition as to what a link in a nonmonotonic multiple inheritance network means and how the inheritance principle, namely that subclass should override superclass, should be interpreted. This prompted interest in formalizing inheritance reasoning, especially for nonmonotonic multiple inheritance systems. In this thesis, we have provided a survey on the different attempts to formalize inheritance reasoning. The second problem with nonmonotonic multiple inheritance systems is that parallel processing in such systems is difficult. The usefulness of nonmonotonic inheritance systems combined with the efficiency and simplicity of the PMPM have prompted interests in using the PMPM to do parallel inference in nonmonotonic multiple inheritance systems. However, the result of such a combination is dispointing. It is soon discovered that simple parallel algorithm running on the PMPM, such as the shortest path algorithm, is not adequate for inheritance reasoning. Even in some simple inheritance networks, the shortest path algorithm may produce incorrect answer.

Despite this shortcoming of the PMPM, marker propagation as a mechanism for parallel inferencing is still attractive because of its simplicity. There have been a number of attempts to circumvent the situation. We have provided a survey upon these attempts. The common characteristic of these attempts is that they start with the PMPM and then try to find ways to do parallel processing or quasi-parallel processing for inheritance networks. In this thesis, we follow a different approach. Since the problems with the PMPM to deal with inheritance reasoning is its oversimplicity, we try to investigate if we can enhance the PMPM so as to make it more suitable for inheritance reasoning. In this thesis, we have suggested two different enhancements. In the first one, marker-value pairs are passed around instead of just markers. Such an enhancement makes it possible to have a completely parallel inheritance algorithm for unambiguous networks. In the second enhancement, a set of link-checking and link-setting commands are provided so that links can be marked with link markers and they can be checked if they are marked with particular markers. Using such an enhanced PMPM, a serial-parallel algorithm is presented which reasons according to the definition of [Touretzky 86]. Arguments for the correctness of the algorithms are also provided.

6.2. Results of the Thesis and Future Research

As for the open problem whether there are natural classes of inheritance networks which admit parallel inference algorithms, this thesis has provided a partial answer. Since the class of unambiguous network is a natural class of inheritance networks, we have demonstrated that parallel inference algorithm is possible, provided that the PMPM is enhanced as suggested.

As for general inheritance networks, we have demonstrated that a serial-parallel algorithm is possible, given that a set of link-setting and link-checking commands are added to the PMPM. We have also analyzed if it is possible to pass around some additional information so that totally parallel algorithms can be designed. But from our analysis, it seems that it is extremely difficult, considering the demand on the capability of the processing elements to analyze the additional information.

The two approaches used in dealing with unambiguous and general inheritance networks are different. For unambiguous networks, additional information is passed around. For general networks, information is deposited onto the links. It is worthwhile to investigate if it is possible to combine the two approaches in order to have a more efficient algorithm for general inheritance networks.

In this thesis, we have distinguish between an unambiguous and a general inheritance network. This is a coarse characterization. As it turns out, the algorithm for general inheritance networks is quite complex. A natural question is whether we can further divide the class of general inheritance networks so that less complex algorithms can be designed.

References

[Bobrow, Winograd 79] Bobrow, D. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language," Cognitive Science 1 (1979), pp. 3-48.

[Brachman 85] Brachman, R., "'I Lied about the Trees' Or, Defaults and Definitions in Knowledge Representation," *The AI Magazine*, Fall 1985, pp. 80-93.

[Brachman, Schmolze 85] Brachman, R. and Schmolze J., "An Overview of the KL-ONE Knowledge Representation System," Cognitive Science 9 (1985), pp. 171-216.

[Carlson 82] Carlson, G., "Generic Terms and Generic Sentences," *Journal of Philosophical Logic 11 (1982)*, pp. 145-181.

[Etherington, Reiter 83] Etherington, D. W. and Reiter, R., "On Inheritance Hierarchies With Exceptions," *AAAI-83*, pp. 104-108.

[Etherington 82] Etherington, D. W., "Finite Default Theories," M.Sc. thesis, Department of Computer Science, University of British Columbia, 1982.

[Etherington 87a] Etherington, D. W., "Formalizing Nonmonotonic Reasoning Systems," *Artificial Intelligence 31 (1987)*, pp. 41-85.

[Etherington 87b] Etherington, D. W., "More On Inheritance Hierarchies with Exceptions: Default Theories and Inferential Distance," *AAAI-87*, pp. 352-357.

[Fahlman 79] Fahlman, S., "A System for Representing and Using Real-World Knowledge," MIT Press, Cambridge, MA, 1979.

[Fahlman, Sejnowski 83] Fahlman, S., Sejnowski, T., "Massively Parallel Architectures For AI: NETL, Thistle, and Boltzmann Machines," *AAAI-83*, pp. 486-490.

[Fahlman et al 81] Fahlman, S., Touretzky, D. and Van Roggen, W., "Cancellation in a Parallel Semantic Network," *IJCAI-81*, pp. 257-263.

[Hayes 79] Hayes, P., "A Logic of Frames," in *Frame Conceptions and Text Understanding*, Walter de Gruyter and Co., 1979.

[Horty et al 87] Horty, J., Thomason, R. and Touretzky, D., "A Skeptical Theory of Inheritance in Nonmonotonic Semantic Networks," Carnegie Mellon University, Computer Science Technical Report CMU-CS-87-175, October 1987.

[Hwang, Briggs 85] Hwang, K. and Briggs, F., "Computer Architecture and Parallel Processing," McGraw Hall Book Company, 1985.

[McCarthy 80] McCarthy, J., "Circumscription - A Form of Non-Monotonic Reasoning," *Artificial Intelligence 13 (1980)*, pp. 27-39.

[McCarthy 86] McCarthy, J., "Applications of Circumscription to Formalizing Common-Sense Knowledge," *Artificial Intelligence 28 (1986)*, pp. 86-116.

[McDermott, Doyle 80] McDermott, D. and Doyle J., "Non-Monotonic Logic I," *Artificial Intelligence 13 (1980)*, pp. 41-72.

[Poole 85] Poole, D., "On the Comparison of Theories: Preferring the Most Specific Explanation," *IJCAI-85*, pp. 144-147.

[Reiter 78] Reiter, R., "On reasoning by default," *Proceedings of the Second Symposium on Theoretical Issues in Natural Language Processing 1978*, pp. 25-27.

[Reiter 80] Reiter, R., "A Logic for Default Reasoning," *Artificial Intelligence 13 (1980)*, pp. 81-132.

[Reiter, Criscuolo 81] Reiter, R. and Criscuolo G., "On Interacting Defaults," *IJCAI-81*, pp. 270-276.

[Rich 83] Rich, E., "Default Reasoning as Likelihood Reasoning," *AAAI-83*, pp. 348-351.

[Sandewall 86] Sandewall, E., "Nonmonotonic Inference Rules of Multiple Inheritance with Exceptions," *Proceedings of the IEEE, vol. 74, no. 10, Oct. 1986*, pp. 1345-1353.

[Touretzky 84] Touretzky, D., "Implicit Ordering of Defaults in Inheritance Systems," *AAAI-84*, pp. 322-325.

[Touretzky 86] Touretzky, D., "The Mathematics of Inheritance Systems," *Research Notes in Artificial Intelligence*, Morgan Kaufmann, 1986.

[Touretzky et al 87] Touretzky D., Horty J. and Thomason R., "A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems," *IJCAI-87*, pp. 476-482.