Boolean Classes

David McAllester
Ramin Zabih

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
Cambridge, Mass. 02139
(617) 253-8827

Abstract

We extend the notion of class so that any Boolean combinations of classes is also a class. Boolean classes allow greater precision and conciseness in naming the class of objects governed a particular method. A class can be viewed as a predicate which is either true or false of any given object. Unlike predicates however classes have an inheritance hierarchy which is known at compile time. Boolean classes extend the notion of class, making classes more like predicates, while preserving the compile time computable inheritance hierarchy.

Introduction

Object-oriented programming languages such as SmallTalk [2], Flavors [3] and CommonLoops [1] all involve the notions of *class* and *object*. A given object can be "in" a class C and thus "inherit" information attached to C. In this paper we view classes as predicates; if an object x is in a class C then we say that C is *true of* x; if x is not in C then we say that C is *false of* x. In Flavors and in CommonLoops a class C is true of x just in case the expression (typep x C) evaluates to T. Unlike ordinary lambda predicates, however, the classes of object-oriented programming languages have a computable inheritance relation. Given any two classes C_1 and C_2 there is a uniform way to determine if C_1 inherits from C_2, i.e. if the fact that C_1 is true of y implies that C_2 is true of y.

Object-oriented programs specify behavior in terms of classes. A behavioral specification, such as a methods or an instance variable, is associated with a particular class C; if C is true of an object x then x should satisfy the specifications associated with C. This paper does not discuss the nature of behavioral specifications or the way that specifications are combined when several specifications apply to the same object. Instead, this paper discusses the nature of classes themselves; Boolean class expressions allow greater precision in naming the class governed by a particular behavioral specification.

Boolean classes are constructed from other classes using :AND's, :OR's, :NOT's. For example, consider a battlefield video-game with the classes TARGET and PROJECTILE where targets are things like tanks, ships and planes while projectiles are things like missiles, torpedoes, or cannon shells. Suppose that the game also has the classes LAND-OBJECT, AIR-OBJECT and SEA-OBJECT. Now consider a particular procedure for displaying the explosion of a surface target, i.e. a target which is not an air object. The class governed by the explosion display procedure can be written as

```
(:AND TARGET (:NOT AIR-OBJECT))
```

The above class need not be mentioned in any user defined inheritance relationships; Boolean classes are automatically placed at the appropriate location in the inheritance hierarchy.

Boolean Classes vs. Simple Multiple Inheritance

Boolean classes make it easier to specify and maintain inheritance hierarchies. In a certain sense any object-oriented language with multiple inheritance can simulate Boolean classes; any inheritance hierarchy involving Boolean classes can expressed as a hierarchy among non-Boolean classes (see below). With Boolean classes however the construction of this inheritance hierarchy can be largely automated and the hierarchy can be automatically updated in non-trivial ways when new classes are introduced and when new inheritance relations are given by the user.

Boolean classes can be simulated in any system with multiple inheritance by manually installing the appropriate inheritance relations. For example consider the Boolean class (:ORC1 C2). This class corresponds to the set of all things which are either in class C1 or in class C2. The Boolean class (:OR C1 C2) can be simulated with a "primitive" class OR-C1-C2 where the user specifies that both C1 and C2 inherit from the class OR-C1-C2. Once this is done any class which inherits from either C1 or C2 will necessarily inherit from OR-C1-C2. Similarly, consider the class (:AND C1 C2) which corresponds to the set of all things which are in both the class C1 and the class C2. This class can be simulated with a primitive class AND-C1-C2 where for each class C3 that inherits from both C1 and C2 the user specifies that C3 also inherits from AND-C1-C2.

Note that simulating the class (:AND C1 C2) requires the user to know about all classes that inherit from both C1 and C2. This is an unreasonable requirement for large systems where different classes are constructed at different times by different users. With true Boolean classes the inheritance relationships for the class (:AND C1 C2) are constructed automatically by the system so the user need not know all the classes that inherit from both C1 and C2.

Technical Difficulties

There are two technical difficulties involving Boolean classes: the uninstantiable class problem and the modularity breakdown problem. The uninstantiable class problem involves classes which are inherently uninstantiable. For example the class (:OR C1 C2) can not be directly instantiated because any direct instantiation of (:OR C1 C2) would yield an object not in either C1 or C2 and thus not in (:OR C1 C2). We solve the uninstantiable class problem by identifying those classes which are inherently uninstantiable and specifying that any attempt to directly instantiate an uninstantiable class should will an error. Modularity breakdown occurs when a class written for one system is inherited by a class in a totally unrelated system. For example consider the class (:NOTC) which includes all objects which are not in class C. The class (:NOTC), and any behavioral specification associated with the class (:NOT C), will be inherited by a totally unrelated class D. The modularity breakdown problem can be solved with simple restrictions on inheritance specifications and on the classes which are allowed to carry behavioral specifications.

This paper begins with a formal specification for a class system with Boolean classes. We then give an algorithm for constructing inheritance relations and identifying uninstantiable classes and present a rigorous proof of the correctness of this procedure. We also formally prove that the modularity breakdown problem is solved via certain simple restrictions on the inheritance hierarchy and on the classes which are allowed to carry behavioral specifications.

Formal Specification

We assume that classes are named with *class expressions* where a class expression is either a class symbol or a Boolean combination of other class expressions.

> **Definition:** A *class expression* is either a class symbol or an expression of the form (:not E), (:or $E_1 \dots E_n$) or (:and $E_1 \dots E_n$) where $E, E_1, \dots E_n$ are other class expressions. A *literal* is either a class symbol or the negation of a class symbol.

We view classes as predicates: given a class C and an object x, if x is in the class C then we say that C is true of x and if x is not in C then we say that C is false of x. Boolean class expressions are interpreted as predicates in the obvious way: the class expression (:not E) is true of an object x just in case E is false of x; (:or E_1 $E_2 \dots E_n$) is true of x just in case some E_i is true of x, and (:and E_1 $E_2 \dots E_n$) is true of x just in case all of the classes E_i are true of x.

In defining a set of classes a programmer provides a set of class expressions and a specified inheritance hierarchy for those class expressions. In languages such as Flavors and CommonLoops the inheritance hierarchy is specified by associating each class with a list of "superior" or "included" classes. For theoretical generality and conceptual simplicity we generalize inheritance specifications to allow for an arbitrary set of *implications* of a certain form.

> **Definition:** An *inheritance specification* is a finite set of implications of the form (:implies C E) where C is a class symbol and E is any class expression.

Intuitively an implication of the form (:implies C E) says that for any object x, if C is true of x the E should be true of x, i.e. every instance of C should be an instance of E, i.e. C inherits from E.

The standard specifications of inheritance relations can be easily translated into implications of the above form. More specifically, to state that a class C inherits from the "components" $C_1, C_2 \dots C_n$, one uses the implications

(:implies C C_1)
(:implies C C_2)
$\vdots$
(:implies C C_n)

If a class C inherits from a class C' then we say that C' is a *generalization* of C. Given an inheritance specification I and a class C we would like to construct the set of generalizations of the class C, i.e. the set of all class symbols which C inherits from.† In languages such as Flavors and CommonLoops the process of computing all of the generalizations of a given class is basically a transitive closure operation: one finds all immediate generalizations, the immediate generalizations of those generalizations, and so on.

Unfortunately computing the set of generalizations of a class is more complex when Boolean expressions are used in the inheritance specification. For example consider the following inheritance specification:

(:implies C_1 (:or C_2 C_3))
(:implies C_2 C_4)
(:implies C_3 C_4)

Let x be an arbitrary object such that C_1 is true of x. The first implication says that either C_2 or C_3 is true of x. In either case the above implications state that C_4 must be true of x. In short, if C_1 is true of an object then C_4 is true of that object. Thus C_4 is a generalization of C_1.

An inheritance specification I can be viewed as a set of formulas of propositional logic. To compute the set of generalizations of a class C one must examine the logical consequences of the formulas in I.

> **Definition:** Let C be a class symbol and let E be a class expression. We say that E is a *generalization* of C under an inheritance specification I (or that C *inherits from* E under I) if the expression (:implies C E) is a logical consequence of the conjunction of all the implications in I.

The above definition treats the inheritance specification I as a formula of Boolean logic (the conjunction of all the implications in I). It is natural to ask whether any Boolean formula could be used as an inheritance specification. The answer is no; to avoid the modularity breakdown problem we have intentionally restricted the inheritance hierarchy to be a set of implications of a certain form. However the restrictions on the inheritance hierarchy are extremely weak; most Boolean formulas can be faithfully translated into a legal inheritance inheritance hierarchy and there is a simple semantic characterization of those formulas which can be translated into a legal inheritance specification.

†For now we consider only those generalizations which are class symbols; Boolean generalizations will be discussed later.

The semantic characterization of the formulas which can be translated into inheritance specifications involves the notion of a "lost" object. We say that an object x is lost if every class symbol is false of x. Note that if I is a legal inheritance specification and x is a lost object then x satisfies every implication in I; x satisfies an implication of the form (:implies C E) because the antecedent C is false of x. Since lost objects satisfy every implication in any inheritance specification, every inheritance specification has a model and thus every inheritance specification is logically consistent (one can never derive a contradiction from the formulas in an inheritance specification.) It turns out that any Boolean formula which is satisfied by lost objects can be translated into a legal inheritance specification.

Lemma I. Let B be any Boolean expressions. If B is true of lost objects then there is an inheritance specification I such that the conjunction of all implications in I is logically equivalent to B.

Proof: Suppose that B is true of lost objects and let E be the conjunctive normal form of B. The expression E is a conjunction of disjunctions of literals where every disjunction of literals. Note that each disjunction in E must contain at least one negative literal because a disjunction which contains only positive literals is false on lost objects and if E contained such a disjunction then E would be false on lost objects violating the assumption that B is true on lost objects. Since every disjunction in E contains a negative literal, every disjunction can be written as (:or (:not C) $L_1 \ldots L_n$) which is logically equivalent to the implication (:implies C (:or $L_1 \ldots L_n$)). (Actually the disjunctive clause might contain only the single literal (:not C) in which case it is equivalent to the implication (:implies C (:not C)).) Thus E is equivalent to a conjunction of implications of the desired form.

The requirement that lost objects satisfy inheritance specifications plays an important role in the solution of the modularity breakdown problem.

Making Instances of a Class

In most object-oriented programming languages objects are created by "instantiating" classes. If the object x was created by instantiating some class symbol C then we will call x an *instantiation* of C. In this section we only consider instantiations of class symbols. We do not allow a class expression E to be instantiated directly. Instead, one can construct the inheritance specification (:implies C E) for some new class symbol C and then one can instantiate C.

Class expressions introduce some subtleties in instantiation. In most object-oriented languages, if x is an instantiation of a class C and E any other class then E is true of x just in case C inherits from E. Unfortunately this principle does not hold when E can be a Boolean class expression. In particular it is possible that C inherit from neither E nor (:NOT E) but clearly either E or (:NOT E) must be true of x. If x is an instantiation of C we must be careful to specify exactly which classes are true of x. It suffices to specify the class symbols which contain x; to determine if a Boolean expression E is true of x it suffices to know whether or not each class symbol in E is true of x.

Specification: Let C be a class symbol, I be an inheritance specification, and let x be an instantiation of C. For any *class symbol* C' we specify that C' is true of x just in case C' is a generalization of C under I.

Now consider the case where C does not inherit from either the class symbol C' or (:not C'). If x is an instantiation of C then the above specification requires that C' is false of x. This implies that (:not C') is true of x even though (:not C') is not a generalization of C.

Uninstantiable Classes

This section formally defines the notion of an uninstantiable class. A procedure for identifying uninstantiable classes is given in a later section. A class is uninstantiable if instantiations of that class would be "pathological". Any attempt to instantiate an uninstantiable class should generate an error.

Definition: We say that an object x *violates* an inheritance implication (:implies C E) if C is true of x but E is false of x. Now let I be any inheritance specification and let C be any class symbol. We say that C is *uninstantiable* under I if an instantiation x of C would violate some implication in I.

It is easy to see how uninstantiable classes arise. For example suppose that I includes both the implications

(:implies C_1 C_2)

and

(:implies C_1 (:not C_2))

Clearly any instantiation of C_1would violate one of these implications.

A more interesting example of an uninstantiable class involves an inheritance specification containing the following single implication:

(:implies C_1 (:or C_2 C_3))

Under this inheritance specification neither C_2 nor C_3 is a generalization of C_1. Both C_2 and C_3 would be false of an instantiation x of C_1. Thus an instantiation of C_1 would violate the above inheritance specification which says if C_1 is true of x then either C_2 or C_3 must be true of x.

Inheriting Methods and Instance Variables

We allow information to be attached to any class expression. For example we might define a method that handles messages sent to objects in the class (:or C_1 C_2). Similarly one might declare that objects in a given class, such as (:and C_1 (:not C_2)), should all have a certain instance variable. In general we will simply speak of "information" that is inherited by objects in a given class. We assume that there is a finite set of *information bearing* class expressions, i.e. class expressions which either have method definitions or instance variables associated with them.

Let C be a class symbol which is instantiable relative to an inheritance specification I. If x is an instantiation of C then the above specifications determine the set of information bearing class expressions which are true of x. The same set of information bearing classes applies to all instantiations of C so it is possible to build a "method table" for the class C which summarizes all the information which

applies to instantiations of C (we are not concerned here with how information gets combined).

To solve the modularity breakdown problem we place a simple restriction on the class expressions that are allowed to carry information. Recall that an object x is called *lost* if every class symbol C is false of x.

> **Specification:** All information bearing class expressions must be false of lost objects, i.e. lost objects do not inherit any information.

For example the class (:NOT AIR-OBJECT) should not carry information because this class is true of lost objects. If the class (:NOT AIR-OBJECT) carried information then that information would be inherited by classes in totally unrelated systems; one would be faced with modularity breakdown.

Any attempt to associate information with a class expression that is true of lost objects should generate an error. The following lemma establishes that the above condition on information bearing classes together with the definition of a legal inheritance specification solves the modularity breakdown problem.

> **Lemma II.** Let I be an inheritance specification and let D be a set of information bearing class expressions. If C is a class symbol which does not appear in either I or D and if x is an instantiation of C then all information bearing class expressions in D are false of x, i.e. x does not inherit any information.

> *Proof:* First we prove that C is the only class symbol that is is true of x. It is sufficient to show that for every other class symbol C', C' is not a generalization of C. To show that C' is not a generalization of C we must show that there exists a propositional model of inheritance specification I which makes C true and C' false. Let y be an object such that C is true of y but no other class symbol is true of y. In particular every antecedent of every implication in I is false of y so y is a model of I which makes C true and C' false.
>
> Now since C is the only class symbol which is true on an instantiation x of C, and since C does not appear in any information bearing class expression, an information bearing class expression E is true on x just in case E is true on lost objects. But since no information bearing class is true on lost objects, no information bearing class is true on x.

Implementation

Unfortunately it can be difficult to compute the set of generalizations of a given class under a given inheritance specification; the algorithm presented here has an exponential worst case running time and we can not expect to find a non-exponential procedure. However the procedure presented here is exponential in the number of "complex" implications in the inheritance hierarchy and in practice only very small fraction of the implications are complex. Furthermore the procedure is modular: the time required to find all generalizations of a given class is not effected by the presence of unrelated classes and inheritance specifications. Thus we expect that the exponential worst case behavior will not be a problem in practice. First we show that the problem of determining whether one class inherits from another is Co-NP compete and thus we can not expect to find a non-exponential algorithm.

Determining Inheritance is CO-NP Complete

It is easy to show that determining whether or not a class symbol C inherits from another class C' under a specification I is *coNP*-complete. More specifically one can reduce the problem of showing that a set of disjunctive clauses is unsatisfiable to the problem of determining whether the implication (:implies C C') follows from an inheritance specification I. Given a set B of disjunctive clauses let C, C' and C'' be symbols not occurring in B and let I be the inheritance specification containing the implication (:implies C (:or C'' C')) together with all implications of the form (:implies C'' E) where E is a clause in B. We will show that C' follows from I and C just in case B is unsatisfiable. If B is unsatisfiable then I implies (:not C'') and thus I and C imply C'. On the other hand if B is satisfiable then consider a model of B in which C'' is true, C is true, and C' is false. This model satisfies all implications in I while making C true and C' false. Thus if B is satisfiable then C' does not follow from I and C.

Computing Inheritance

The system of Boolean classes described here has not yet been implemented. However there is a relatively simple algorithm for determining whether a class is instantiable and for determining the set of information bearing classes that are generalisations of a given instantiable class. The first step in this algorithm is to convert the inheritance specification I into a canonical form.

> **Definition:** An inheritance specification I is said to be in *conjunctive normal form* if every implication in I has the form (:implies C (:or L_1 L_2 ...)) where each L_i is a literal.

We allow the disjunction in the consequent of an implication to contain only a single literal, in which case the implication can be written as (:implies C L). Any inheritance specification I can be converted to an expression in conjunctive normal form. To see this recall that an inheritance specification I consists of implications of the form (:implies C E) where C is a class symbol and E is a class expression. The class expression E can be written in conjunctive normal form, i.e. E can be written as:

(:and (:or $L_{1,1}$ ... $L_{1,n}$) (:or $L_{2,1}$ ... $L_{2,m}$) ...)

The implication (:implies C E) can then be written as a set of implications of the form

(:implies C (:or $L_{j,1}$ ... $L_{j,n}$))

Of course converting an expression to conjunctive normal form requires an exponential amount of work in general. However it seems unlikely that this would be a problem in practice; the Boolean expressions involved should usually be given in conjunctive normal form anyway.

Given a set an inheritance specification I we are now interested in determining which class symbols are instantiable and for each instantiable symbol C we are interested in determining the set of class symbols which generalize C. To do this we assume that the inheritance specification I has been converted to conjunctive normal form. Implications of the form (:implies C L) will be called *simple* while implications of the form (:implies C (:or L_1 $L_2 \ldots L_n$)) (for $n > 1$) will be called *complex*. Current object-oriented programming languages only allow for simple inheritance implications and these are indeed the easiest to process.

Definition: Let T be any set of literals and let I be an inheritance specification in conjunctive normal form. The *simple closure* of T with respect to I is the least set of literals T' containing T such that if C is a class symbol in T' and (:implies C L) is a simple implication in I, then L is in T'.

To compute the simple closure of a set of literals T it is sufficient to compute the transitive closure of the directed graph given by the simple implications in I. The details of this computation are left to the reader. Now let C be a class symbol. Clearly every literal in the simple closure of the singleton set $\{C\}$ is provable from C and I. However, since I may contain complex implications there may be symbols which follow from C and I but which are not in the simple closure of $\{C\}$. To find all symbols which follow from C we must enumerate models of C. A symbol C' follows from C just in case C' is true in every model M of I such that C is true in M.

A model of propositional formulas (expressions) is usually taken to be a truth function which maps each symbol to either **true** or **false**. However, rather than introduce truth functions on class symbols, we will represent a model by a set M of class symbols; members of M are considered to be true while class symbols not in M are taken to be false.

Definition: Let M be a set of class symbols. We say that a literal L is *true under* M if either L is a symbol in M or L is of the form (:not C) where C is not in M. Let I be an inheritance specification in conjunctive normal form. The set M is called a *model of* I if for every implication of the form (:implies C (:or L_1 $L_2 \ldots L_n$)), if C is in M then one of the literals L_i is true in M.

A symbol C inherits from a symbol C' just in case the implication (:implies C C') is provable from I. But (:implies C C') is provable from I just in case every model of I which contains C also contains C'. More specifically, the intersection of all models of I which contain C yields the set of all class symbols which are generalizations of C. The following procedure enumerates models I which contain C.

The procedure takes one explicit argument T which is a set of literals. The procedure also makes use of the inheritance specification I. We will show that a class symbol C' is a generalizations of C under I just in case C' is a member of every model returned by the following function when applied to the singleton set $\{C\}$.

Function: All-Models(T) takes a set of literals and produces a set of models.

1. [Initialization] Let T' be the simple closure of T with respect to I.

2. [Detect inconsistency] If T' is inconsistent, i.e. if there is some symbol C in T' such that (:not C) is also in T, then return the empty set (there are no models of T).

3. [Construct new model] Let
   (:implies C (:or L_1 $L_1 \ldots L_n$))
   be a complex implication in I such that C is in T' but none of the literals $L_1, L_2, \ldots L_n$ are in T'. If there is no such implication in I then return $\{M\}$ where M is the set of class symbols in T'.

4. [Recurse] If there is such an implication in I then return
   $$\bigcup_{1 \le i \le n} \textbf{All-models}(T' \cup \{L_i\})$$

Note that if there are no complex implications in I then **All-models**($\{C\}$) is either empty or contains exactly one model which is derived by computing the simple closure T' of the singleton set $\{C\}$.

The above procedure can be made more efficient in several ways. The search for a complex implication in step 3 can be optimized to avoid searching all implications in I. Also, the set of literals T used in this procedure can be represented with a hash table so that membership tests take unit time on average. The details of these optimizations are left to the reader. It is important to note, however, that this computation is only exponential in the number of complex implications of the form specified in step 3. All that remains is to show that this algorithm produces all the possible models.

Lemma III. Every element of **All-Models**($\{C\}$) is a model of I which contains C. Furthermore every model of I which contains C also contains (as a subset) some element of **All-Models**($\{C\}$).

The above lemma implies that the intersection of the models in **All-Models**($\{C\}$) is equal to the intersection of all models of I which contain C. In other words this intersection is the set of generalizations of C under I. The proof of this lemma is presented in the appendix.

If the above procedure returns the empty set when applied to the singleton $\{C\}$ then there are no models of I which contain C and thus I implies (:not C) so C is not instantiable. If the procedure returns a set of models then the intersection of those models is the set of class symbols which are generalizations of C. Given the set of symbols

which are generalizations of C one can can consider a hypothetical instantiation y of C. This hypothetical instantiation will satisfy every simple implication in C, but there might be some complex implication which is violated by y. If some implication is violated by y then C is uninstantiable. On the other hand if every implication in I is satisfied by y then C is instantiable and we can compute the set of information bearing class expressions that are true of the hypothetical instantiation y.

Possible Extensions

For any predicate C one must be able to specify the behavior of functions and methods when applied to objects that satisfy the predicate C. This can be done in one of two ways: one can write explicit conditionals in the code for methods and functions or, for certain predicates, one can represent the predicate C as a class and define methods for that class. Boolean classes expand the set of predicates which can be represented as classes. It might be possible to extend the class vocabulary even further so that other predicates can be represented as classes. For example one might want to define the class of ships whose current momentum is greater than 1000, or the class of missiles that are within ten miles of their targets.

Predicates can be divided into three groups. First there are *instance ignoring* predicates. A predicate C is instance ignoring if the truth of C on an object x depends only on the class of x (the class that x is an instantiation of) and not on any particular properties of the instance x. All boolean class expressions are instance ignoring. Second there are *instance sensitive* but *time invariant* predicates. A predicate C is time invariant if the truth of C on an object x does not change over time. Third, there are *time varying* predicates. A predicate C is time varying if the truth of C on an object x changes over time. It is progressively more difficult to extend the class vocabulary to these three types of predicates, because inheritance information is available respectively at compile time, at object creation time and at run time.

As an example of an instance sensitive predicate that is time-invariant consider a class `COUPLING-CAPACITOR` where this class contains those capacitor objects whose capacitance is above a given threshold. We assume that the capacitance of a capacitor is given at object creation time and never changes. Whenever a capacitor object is created one could determine whether or not it is an instance of `COUPLING-CAPACITOR`.

Time varying predicates are quite common and one could imagine specifying methods in terms of classes defined by time varying predicates. It should be possible to implement time varying classes by automatically converting the behavioral specifications associated with classes into run-time conditionals in the code for methods.

The potential benefits and pitfalls of extending the class vocabulary to more general kinds of predicates are not yet clear; we have not investigated the uninstantiable class problem or the modularity breakdown problem for instance sensitive or time varying classes. It seems likely that any implementation of instance sensitive or time varying classes would involve in-line conditional tests in the code for methods. Thus it is not clear that there is any advantage in representing these predicates as classes as opposed to using these predicate in traditional in-line conditionals.

Acknowledgments

Richard Zippel helped persuade us to explore alternative ways of thinking about object-oriented programming. Alan Bawden provided useful comments and insight. The S-1 Project at Lawrence Livermore National Laboratory and Schlumberger Palo Alto Research provided facilities that aided in preparing this paper.

This paper describes research done at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Ramin Zabih is supported by a fellowship from the Fannie and John Hertz Foundation.

Appendix: The Proof of Lemma III

Consider the following procedure given in the third section of this paper.

Function: All-Models(T) takes a set of literals and produces a set of models.

1. [Initialization] Let T' be the simple closure of T with respect to I.

2. [Detect inconsistency] If T' is inconsistent, i.e. if there is some symbol C in T' such that (:not C) is also in T, then return the empty set (there are no models of T).

3. [Construct new model] Let us assume that
   (:implies C (:or L_1 $L_1 \ldots L_n$))
   is a complex implication in I such that C is in T' but none of the literals $L_1, L_2, \ldots L_n$ are in T'. If there is no such implication in I then return $\{M\}$ where M is the set of class symbols in T'.

4. [Recurse] If there is such an implication in I then return
   $$\bigcup_{1 \le i \le n} \textbf{All-models}(T' \cup \{L_i\})$$

Note that a recursive call in step 4 can return the empty set in which case that recursive call does not contribute any models and has no effect on the result. A recursive call returns the empty set if the set of literals passed to that call is inconsistent with the inheritance specification I.

We wish to prove the following lemma:

Lemma III.

(a) Every element of **All-Models**($\{C\}$) is a model of I which contains C.

(b) Every model of I which contains C also contains (as a subset) some element of **All-Models**($\{C\}$).

To prove part (a) let M be a model,

$$M \in \textbf{All-Models}(\{C\})$$

Clearly M contains C. To show that M is a model of I note that M must have been returned at step 3 of some invocation of the procedure. At step 3 of the procedure there exists a consistent set of literals T' such that M is the set of class symbols in T' and for every (simple or complex) implication in I of the form (:implies C' (:or L_1 $L_2 \ldots L_n$)), if C' is in T' then some L_i is in T'. To show that M is a model of I consider an implication of the form (:implies C' (:or L_1 $L_2 \ldots L_n$)). We must show that if C' is in M then some L_i is true in M. If C' is in M then C' is in T'. But this implies that T' contains some L_i. Now if L_i is a positive literal then it is also contained in M and we are done. On the other hand if L_i is a negative literal of the form (:not C'') then since T' is consistent C'' is not in T' and thus not in M so (:not C'') is true in M.

Now we must prove that every model of I which contains C also contains (as a subset) some element of **All-Models**($\{C\}$). This is proven via a more general induction hypothesis on the function **All-Models**.

> **Definition:** Let T be any set of literals and let I be any inheritance specification in conjunctive normal form. A *model M of T* (relative to I) is a model of I such that every literal in T is true in M.

Note that a model M of a set of literals T can contain class symbols which do not appear in T. In the extreme case M might be infinite while T is finite. We will show that, in general, every model of I and T contains (as a subset) some member of **All-Models**(T). In particular this implies that every model of I and $\{C\}$ (i.e. every model of I which contains C) contains (as a subset) some member of **All-Models**($\{C\}$).

To prove the general induction hypothesis we first note that the function **All-Models** must always terminate because the number of literals increases in every recursive call and if the number of literals in T becomes larger than the number of class symbols appearing in I then T must be inconsistent and the procedure terminates.

Now we assume that the induction hypothesis holds for recursive calls and we show that it must then hold for the top level call. First note that if T' is the simple closure of T every model of T (relative to I) is also a model of T'. Thus if T' is inconsistent then there are no models of T and the lemma holds. Furthermore, every model of T is also a model of T' and therefore must contain (as a subset) all of the positive literals in T'. Now suppose the procedure exits in step 3 by returning the positive literals in T'. Since every model of T' must contain (as a subset) the positive literals in T' the lemma holds. Finally suppose that the procedure returns the union computed in step 4. Let M be any model of I and T and let (:implies C (:or L_1 $L_1 \ldots L_n$)) be the implication found in step 3. Since T' contains C, the model M must also contain C. Furthermore, since by definition M is a model of I, some literal L_i must be true in M. This implies that M is a model of $T' \cup \{L_i\}$ (relative to I) for some L_i in the implication. But we have assumed that the induction hypothesis holds for recursive calls and so M contains (as a subset) some member M' of **All-Models**($T' \cup \{L_i\}$). But M' is a member of the union computed at step 4 so the lemma holds.

References

1. D. Bobrow; K. Kahn; M. Stefik; and G. Kiczales "Common Loops" Xerox Palo Alto Research Center (1985)

2. Daniel H. H. Ingalls. "The Smalltalk-76 Programming System: Design and Implementation" *Proceedings of the Principles of Programming Languages Symposium* (1976)

3. Daniel L. Weinreb; and David A. Moon "Lisp Machine Manual" MIT Artificial Intelligence Laboratory (1981)