Published in the Data & Knowledge Engineering 21 (1997) 253-280.

Handling Inheritance in a System Integrating Logic in Objects

Ioannis Hatzilygeroudis*
Dept. of Computer Engineering & Informatics
School of Enginering, University of Patras
26500 Patras, Greece
&
Computer Technology Institute
P.O. Box 1122, 26110 Patras, Greece

email: ihatz@cti.gr
fax: +30-61-991909

Han Reichgelt
Dept. of Computer Science
University of the West Indies
Mona, Kingston 7, Jamaica

email: han@uwimona.edu.jm

Abstract

The inheritance mechanism of SILO, a system integrating a many-sorted logic within an object-based framework, is presented. In order to be adequate for knowledge representation, it comprises two components, a hardwired and a user-definable. Due to use of typed (sorted) terms, a variety of specialisation types between logical formulas (axioms) are introduced and defined. Thus, the hardwired component is able to represent a variety of inheritance/specialisation relations between objects. The notion of a conflict is defined and conflict detection theorems are introduced. Also, consequence retraction is introduced and used alongside attribute/predicate overriding to resolve conflicts. The user-definable component consists of a number of user definable functions, called meta-functions, which are able to implement both global and local inheritance control. It is based on a partial reflection meta-level architecture.

Keywords: knowledge representation, objects, many-sorted logic, integrated system, logical formulas inheritance, specialisation types, inheritance control.

1. Introduction

SILO is a general purpose hybrid knowledge representation language/system integrating logic in objects [10, 11]. A distinguishing feature of SILO is that it gives

* Corresponding author.

pre-eminence to objects, not to logic. A first-order many-sorted logic is used within an object-based framework, where objects are distinguished in classes and instances and are organised in a hierarchy that allows for multiple inheritance. Logic is used to represent the slots and methods associated with an object. Consequently, inheritance becomes a process of inheriting logical expressions from the ancestor(s) of an object. Thus, inheritance remains a fundamental mechanism of SILO, as in most object-based systems.

Most of the systems that combine logic and objects are extensions of the logic programming paradigm, that is they give pre-eminence to logic and consider the combination from the programming point of view rather than that of knowledge representation (e.g. [7, 8, 14, 17]). Typically, these systems amount to implementing an object-oriented programming language in a logic programming language, just as many of the (early) Integrated Artificial Intelligence Programming Environments, such as LOOPS [2] and KEE [6], included an object-oriented programming language implemented in the underlying functional programming language. Since the programming point of view mainly aims at representational rigidity rather than representational flexibility, most of the existing systems have a restricted and fixed inheritance mechanism which is quite inflexible, hence inadequate for knowledge representation. Knowledge representation requires the ability to be declarative and flexible, which is not offered by existing object-oriented approaches. For example, the class-instance model of object-oriented languages is too inflexible to cover all types of specialisation used in knowledge representation [10].

So, an inheritance mechanism, in order to be adequate for knowledge representation, should be able to implement/recognise a wide variety of possible specialisations between a class and its subclasses or a class and its instances. However, there seems to be no hardwired inheritance mechanism that satisfies this requirement in a multiple inheritance system. This is mainly due to difficulties concerning representation of domain-dependent specialisations and determination of the inheritance path [5, 19]. To overcome these difficulties, SILO, apart from a rich

hardwired inheritance component, provides a *user-definable inheritance componen*t for controlling inheritance, using a meta-level architecture.

In this paper, the inheritance mechanism of SILO is mainly presented. The outline of the paper is as follows. Section 2 presents how knowledge is structured in SILO both globally and locally. In Section 3, the basics of the domain knowledge representation in SILO are presented. Section 4 deals with the hardwired inheritance component. Section 5 describes how inheritance control is achieved in the user-definable inheritance component. Section 6 discusses related work and, finally, Section 7 concludes.

2. Structuring Knowledge in SILO

2.1 Object structure

Two types of objects are distinguished in SILO. An *instance-object* (or *instance*) contains knowledge about an individual concept. A *class-object* (or *class*) contains knowledge related to a generic concept. Any object in SILO is described via a set of *attributes/predicates* (see Section 3) and consists of three parts: structure-part, knowledge-part and control-part. The *structure-part* of an object accommodates knowledge about its hierarchical relations as well as its attributes/predicates. The *knowledge-part* of an object includes knowledge for deducing values for its attributes/predicates. While the structure-part and the knowledge-part of an object concern *domain knowledge*, its *control-part* concerns *control knowledge*, that is knowledge about how to use domain knowledge, often called meta-knowledge [1, 28]. Control knowledge refers to both deduction and inheritance.

2.2 The specialisation/inheritance model

Objects in SILO are organised in a *hierarchy* which can be graphically represented as a directed acyclic graph (see e.g. [23 Ch.6] for the basic terminology on graphs) whose nodes represent objects with the object `object` as its root (see Fig.1), in common with most object-oriented languages [19]. Instances are terminal nodes in a hierarchy.

Each class, except `object`, is an immediate subclass of one or more classes higher up, called its immediate superclass(es). The link between a class and an immediate subclass of it represents a specialisation/inheritance relation, called an *immediate-subclass-of* relation. This means that an immediate subclass can differentiate itself from its immediate superclass(es) by a number of ways (see Section 4). An immediate subclass can have new attributes/predicates defined in it. We further define the *subclass-of* relation as the transitive closure of the 'immediate-subclass-of' relation.

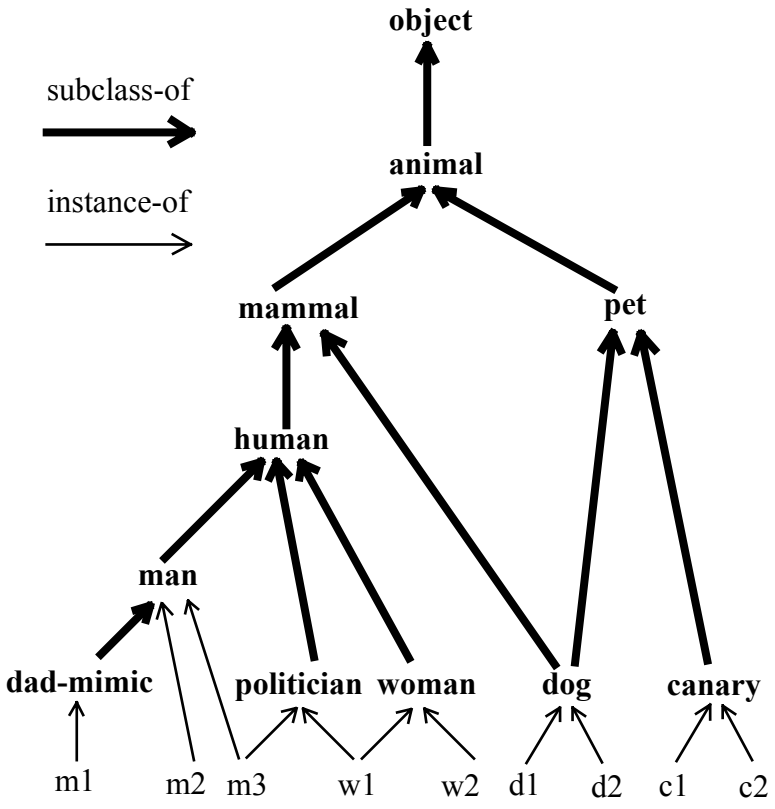

Fig.1 A partial SILO hierarchy

A class, apart from subclasses, can also have instances attached to it (e.g. `man` Fig.1). Classes that have only instances attached to them are called *terminal classes* (e.g. `dog` in Fig.1). There is no terminal class with no instances attached to it. An instance may belong to more than one class. The link between a class and an instance

of it represents a *restricted* specialisation/inheritance relation, called an *instance-of* relation. It is restricted in the sense that an instance cannot have new attributes/predicates defined within itself. Furthermore, instances cannot be further specialised.

So, the specialisation/inheritance model of SILO is more flexible than that of standard class-based systems, which is based on the set theory, and a bit less flexible than that of standard frame-based systems, which is based on the prototype theory (see [19 Ch.7]). For example, standard class-based systems do not allow for new methods to be defined within instances, whereas SILO does. On the other hand, standard frame-based systems allow instances to have new slots (attributes) defined in them, whereas SILO doesn't. However, SILO retains the declarative structure of frame-based systems. Therefore, it could be said that SILO's specialisation/inheritance model is based on a *restricted prototype theory*.

Any SILO hierarchy is defined by the user, by explicitly declaring the 'immediate-subclass-of' and the 'instance-of' relations between objects. In Fig.1, an incomplete[1] hierarchy of objects, with the links to be defined by the user, is depicted.

2.3 Hierarchy issues

We use C_i and O_i to represent class and instance symbols respectively. We also use "<<" and "<" to represent the 'subclass-of' and the 'instance-of' relations respectively. So, $C_j << C_i$ means that C_j is a subclass of C_i or equivalently that C_i is a superclass of C_j. For example, in the hierarchy of Fig.1, `human` << `mammal`, `mammal` << `animal` and `human` << `animal`; `mammal` is an immediate subclass of `animal` too. Also, $O_j < C_i$ means that O_j is an instance of C_i or equivalently that C_i is a class of O_j. For example, `m3` < `man`, `m3` < `politician` and `w2` < `woman`.

In a hierarchy, a subgraph that has a class C_i as its root is called the C_i *class graph* (or simply the C_i *graph*). The set of the instances that belong to the C_i graph is called the *potential* of C_i, represented by $I(C_i)$. Obviously, each class in the C_i graph is a

[1] "incomplete" and "partial" mean that there are other objects (classes or instances) not depicted in the figure, for the sake of simplicity.

subclass of C_i and the corresponding class graph a subgraph of the C_i graph. Two classes C_i, C_j are (declared to be) *disjoint*, if they have no common instances, that is $I(C_i) \cap I(C_j) = \varnothing$.

For example, in Fig.1, the potential of `human` is $I(human)$ = {`m1`, `m2`, `m3`, `w1`, `w2`}. Also, the `mammal` and `pet` class graphs are subgraphs of the `animal` class graph. `man` has two instances, {`m2`, `m3`}, and a subclass, `dad-mimic`. Furthermore, `man` and `woman` are (declared to be) disjoint.

The following *hypothesis* is considered to be true of any (complete) SILO hierarchy:

> There are no two classes belonging to different branches of a class graph such that the potential of the one is a subset of the potential of the other.

This is a conceptual requirement necessary for proving Lemma1 below, which is in turn necessary for proving Lemma 2 (Section 4.2.1) and Theorem 1 (Section 4.3.2).

Lemma 1. $I(C_k) \subseteq I(C_m)$ iff $C_k << C_m$.

Proof.

<= Since $C_k << C_m$, C_k lies on a branch of the C_m class graph, hence the C_k graph is a subgraph of the C_m graph. Consequently, the potential of the C_k graph is a subset of the potential of the C_m graph, that is $I(C_k) \subseteq I(C_m)$.

=> Since $I(C_k) \subseteq I(C_m)$, C_k is on the same branch as C_m (above hypothesis). Consequently, $C_k << C_m$.

The practical implication of Lemma 1 and the above hypothesis is that there should not be an implicit (: not declared explicitly by the user) subclass of a class on a different branch. This is, of course, responsibility of the user, since detection of such a situation by the system would be impractical.

3. Domain Knowledge Representation

An integrated language is used for the description of the domain knowledge in the structure-part and the knowledge-part of an object that can be considered as a form of a *many-sorted logic* [4]. It consists of two component languages the basics of which are described in the following (for a more detailed description see [11, 12]).

3.1 Structure Declaring Language

The one component language, called *structure declaring language* (SDL), concerns the structure-part of an object and consists of two main types of declarations. The first, *link declarations*, comprises declarations of the object's (super)classes, subclasses and/or instances as well as declarations of which sibling classes of the (class) object are disjoint with it.

The second, *attribute declarations*, represent restrictions on the values of the object's attributes.

> **Definition 1** (Attribute declaration). An *attribute declaration* is an expression of the form $((p_i^n\ C_1\ \ldots\ C_n)\ (n_1\ n_2))$, where p_i^n is an n-place attribute symbol ($n \geq 0$), $C_1,\ldots,C_n$ are class symbols and n_1, n_2 are optional positive integers.

In Definition 1, class symbols represent the types of the corresponding component values (see below) of the attribute, while the two integers represent the minimum and maximum number of values the attribute can take. If one of n_1, n_2 is to be left unspecified, an '!' is used in its place.

In general, an attribute is an *n-place attribute* ($n \geq 0$), that is a value of it is an n-tuple consisting of *n component values*. If $n = 0$ it is a *degenerate attribute*, that is an attribute with no value, if $n = 1$ it is a *simple attribute*, otherwise it is a *composite attribute*. Attributes are distinguished in single-valued and multi-valued attributes. An attribute is *single-valued* if it is allowed to take only one n-tuple as its value, otherwise it is *multi-valued*. Obviously, $n_1 = n_2 = 1$ declares a single-valued attribute.

For example, the declaration "((member man) (2 5))" in class `small-team` concerns the multi-valued attribute 'member' and denotes that "the members of a small team are at least two, at most five and are men".

SDL corresponds to the description of the signature(s) in a many-sorted logic [4].

3.2 Message Passing Logic

The other component language is used for the description of the knowledge-part of an object and is a variant of classical first-order predicate calculus (FOPC), called

message passing logic (MPL). Cambridge Polish notation is used with as connectives {~ , & , V , =>} and quantifiers {forall , exists} (see the definitions below and also [9 Ch.2, 24 Ch.3] for the basic terminology of FOPC).

There are two prime types of terms in MPL, namely constants and variables[2]. A *constant* is an instance symbol, that represents an individual concept. Variables are actually *typed* (or *sorted*) variables, that is variables whose range of values is restricted either explicitly or implicitly. We use v_i to represent variable symbols. In MPL, a variable symbol has '?' as its first character (e.g. ?x). An *explicitly typed variable* has the form

$$v_i{:}C_i$$

where the class symbol C_i represents its type. Obviously, the range of values of v_i is equal to the potential of C_i, $I(C_i)$. An *implicitly typed variable* v_i is considered to be either of the same type as the explicitly typed variable with the same symbol in the same formula, if any, or of type 'object'. Finally, there is the *special variable* '?self ' which has a special semantics, specified in the subsequent sections.

Definition 2. (MPL term). An MPL *term* is either a constant or a variable (implicitly or explicitly typed) or the special variable '?self '.

Definition 3. (MPL atom). An MPL *atomic formula* (or *atom*) is an expression of the form (p_i^{n+1} t_1 ... t_n t_o), where p_i^{n+1} is an *(n+1)*-place predicate symbol ($n \geq 0$) and t_1,..., t_n, t_o its arguments, that are terms.

The last argument t_o always denotes the object the atom refers to and is called the *object argument*. The predicate symbol p_i^{n+1} represents an attribute of the object denoted by the object argument. That is, for any *n*-place attribute there is a homonymous *(n+1)*-place predicate and vice versa. The terms t_1, ... , t_n represent component values of the predicate's homonymous attribute and are called *value arguments*. In the same way as with attributes, we distinguish between single-valued

[2]There is actually another type of terms, called *evaluable terms*, which however are only used for representation conciseness reasons. MPL formulas containing evaluable terms are internally translated to equivalent ones with only constants and variables (see [12]).

and multi-valued predicates. So, "attribute" and "predicate" are treated as almost identical notions throughout the paper.

MPL does not directly support functions, but it does it indirectly (see [12]).

Definition 4. (MPL formula). A well-formed MPL formula (wff) is defined as follows:

(1) an atom is a wff.
(2) if f is a wff, (~ f) is also a wff.
(3) if f_1 and f_2 are wffs, (V f_1 f_2), (& f_1 f_2) and (=> f_1 f_2) are also wffs.
(4) if f is a wff and v_i is a (free) variable in f, ((forall v_i) f) and ((exists v_i) f) are also wffs.
(5) nothing else is a wff.

A variable is free if it is not in the scope of any quantifier.

3.3 MPL Clausal Form

Because SILO uses a resolution-based reasoner [11, 12], MPL formulas are converted into MPL clauses, called *axioms*. Axioms are stored in the knowledge-part of an object. An axiom a_i is represented as a set of MPL literals $\{l_{i1}, l_{i2}, \ldots, l_{in}\}$ with an implicit disjunction between them. A *literal* is either an atom (*positive literal*) or a negated atom (*negative literal*). A *unit axiom* is an axiom consisting of exactly one literal.

Definition 5 (Self-literal). A literal is a *self-literal* if its object argument is either '?self ' or the symbol of the object it is stored in.

So, a self-literal concerns knowledge related to the object it is stored in.

Due to typed terms, apart from normal *Skolem terms*, *typed Skolem terms* (constants and functions) are also resulted in the MPL clausal form (see [12] for details). An MPL literal is a *ground literal* if its arguments are *ground terms*. Ground terms are: constants, Skolem constants, Skolem functions with ground arguments and the special variable '?self '.

There are two types of axioms in the knowledge-part of an object, namely slot-axioms and method-axioms. A *slot-axiom* is a unit axiom consisting of a self-literal. Any other axiom is a *method-axiom*. For example, ((num-of-legs 2 ?self)) in `human`

and ((~ (eats ?x:animal-product john))) in `john` are slot-axioms, whereas ((lives-in ?x ?self) (~ (works-in ?x:city ?self))) in `human` and ((lives-in ?y john) (~ (father ?x john)) (~ (lives-in ?y:city ?x))) in `john` are method-axioms. Slot-axioms correspond to slot-value pairs and method-axioms to methods of standard class-based languages or if-needed procedures of standard frame-based languages.

Any axiom stored in an object refers to attributes/predicates of the object. Thus, we introduce the following definitions, where, for the sake of simplicity, we omit arity numbers from attribute/predicate symbols.

> **Definition 6a** (Positive Reference/Axiom). An axiom *a positively refers* to an attribute/predicate *p*, denoted by $a[p]^+$, if it contains at least a positive self-literal of the attribute/predicate. Axiom *a* is called a *positive axiom*.

> **Definition 6b** (Negative Reference/Axiom). An axiom *a negatively refers* to an attribute/predicate *p*, denoted by $a[p]^-$, if all its self-literals are negative literals of the attribute/predicate. Axiom *a* is called a *negative axiom*.

To express the fact that an axiom *a* is either positive or negative, we use $a[p]$.

In SILO, an *attribute/predicate definition* consists of an attribute declaration and one or more axioms that refer to that attribute/predicate. Each axiom represents one or more *attribute/predicate values* either positive or negative, depending on whether it is positive or negative. For example, ((eats ?x:vegetable ?self)) and ((lives-in ?x ?self) (~ (works-in ?x:city ?self))) represent a number of *positive values* for 'eats' and 'lives-in', whereas the negative axioms ((~ (eats ?x:meat ?self))) and ((~ (plays ?x mike)) (~ (likes ?x:leisure-thing mike))) a number of *negative values* for 'eats' and 'plays'. So, one or more positive axioms with the associated attribute declaration constitute a *positive definition* for the attribute/predicate. Correspondingly, one or more negative axioms with the associated attribute declaration constitute a *negative definition* for the attribute/predicate.

3.4 Reasoning and message passing

Reasoning in SILO is closely related to *message passing*. A resolution based reasoning process starts off by sending a message (query) to an object and takes place

in the context of that object. The *context* of an object is defined as its local theory plus the theories it can inherit from objects higher up. The *local theory* $\vdash_{O_i}$ of an object O_i consists of the axioms (clauses) stored in itself. When a query (theorem) is sent to an object, the system first tries to answer (prove) it using the object's local theory. If this fails, then the object extends its theory by successively inheriting axioms from its (super)classes and new proof attempts are made. Due to possible message passings during resolution, an (initial) reasoning process in the context of an object may trigger a number of other reasoning (sub)processes in the contexts of other objects. The object O_c in whose context reasoning is currently taking place is called the *current object*. The available axioms at any time for proving a theorem in the context of O_c constitute the *current theory* S_c. The special variable '?self ' is always bound to the symbol of the current object.

Message passing is achieved via message-literals. A *message-literal* is a literal in a method-axiom whose object argument denotes an object different from the object the axiom is stored in. A message-literal, in contrast to a self-literal, concerns knowledge related to an object different from the object it is stored in. For example, (~ (lives-in ?y ?x)) in the method-axiom ((lives-in ?y john) (~ (father ?x john)) (~ (lives-in ?y ?x))) stored in `john` is a message literal, since its object argument is not "john", but a variable which will be eventually bound to an object symbol, say "paul", that represents the father of John. Message passing to `paul` then means a theorem proving request in the context of `paul` with the literal as the (negated) theorem to be proved. (For a fuller description of reasoning and message passing in SILO see [11, 12]).

4. Hardwired inheritance

4.1 Introductory issues

We distinguish two aspects of inheritance. The first, called *content inheritance*, concerns which part of the domain knowledge of an object is inherited. The second, called *inheritance order*, concerns the order in which the classes/superclasses of an instance or a class with multiple parents are visited for inheritance. We further distinguish two aspects of content inheritance. The first is called *complete inheritance*

and concerns inheritance of the attribute declarations and the axioms themselves. The second is called *atomic inheritance* and concerns inheritance of the atomic consequences of the axioms. Thus, while complete inheritance refers to all the consequences of axioms, atomic inheritance refers to their atomic consequences. Although both aspects of content inheritance are useful to knowledge representation, existing combinations of logic and objects, except [21], address either complete inheritance (e.g. [7, 15, 18]) or atomic inheritance (e.g. [20]), but not both as SILO does.

Apart from various inheritance aspects, there are also a number of knowledge *specialisation types* (or *specialisations*) required for knowledge representation. The more specialisations an inheritance mechanism is able to support the more flexible a system is in representing differentiations between objects. This is a key factor in knowledge representation, in contrast to programming. The specialisation types supported by SILO are outlined below, where by attribute/predicate values we refer to both positive and negative ones.

(a) *addition*: when definition of a new attribute/predicate, i.e. a new attribute declaration and/or axiom(s), is introduced.

(b) *extension*: when the values of a multi-valued attribute/predicate are extended by a number of values.

(b.1) *pure extension*: when the extending values have none in common with the old ones or

(b.2) *impure extension*: when the extending values have common values with the old ones.

(b.2.1) *including extension*: when the extending values are a superset of the old ones.

(b.2.2) *overlapping extension*: when the extending values overlap with the old ones.

(c) *substitution*: when a new definition of an attribute/predicate is introduced as a substitute for the old one.

(d) *refinement*: when the values of a multi-valued attribute/predicate are restricted to a subset of the old ones.

(e) *exception*: when one or more values of an attribute/predicate are excluded.

(e.1) *full exception*: when all of the values are excluded.

(e.2) *partial exception*: when a subset of the values are excluded.

As it is clear, specialisation types refer to attributes/predicates, not to objects. So, a *specialisation/inheritance relation*, e.g. a subclass-of relation, is composed of a variety of specialisations between the attributes/predicates, hence the axioms, of the two involved classes. In the case of an instance-of relation, because of the restricted specialisation employed, addition is not applicable, as instances cannot have new attributes/predicates defined in them. Existing systems do not address all of the above specialisation types. They do not usually provide ways of implementing one or more of extension, refinement and exception of knowledge. For example, the system in [21], although supports complete and atomic inheritance, it does not provide direct ways for representing refinements and partial exceptions.

Typically, in a multiple inheritance system, an instance/class inherits knowledge from all of its classes/superclasses. Multiple inheritance causes no problems at all as long as there is no conflicting knowledge, either within an instance/class and a class/superclass of it or within different classes/superclasses of it. This is the case when e.g. addition or extension of attributes/predicates are only needed. Then, an instance/class inherits all the knowledge from within its classes/superclasses. However, when e.g. substitution and/or refinement and/or exception of knowledge are also required, conflicts may be created and problems arise. In this case, in order to resolve conflicts, the element (e.g. axiom) with more specific information invalidates the one(s) with less specific information. So, the problem is two-fold: how to detect conflicting knowledge and how to determine its most specific occurrence, that is the one residing lower down in the hierarchy.

In the following subsections we address these issues. Our aim is to a) provide formal definitions for the specialisation types specified above and the notion of a conflict, and b) formalise implicit, natural detection rules for conflicts, so that no extra constructs are required in the language.

4.2 Specialisation types

4.2.1 Basic notions

In this subsection, we introduce some notions and a lemma to be used in the next subsection. In the following, by "representation element" we mean either a term t or a literal l or an axiom a. $G(x)$ represents the set of the ground instances (or instantiations) of the representation element x. So, $G(t) = \{t\}$ if t is not a variable, i.e. it is a constant or a Skolem term, and $G(t) = I(C)$ if t is a variable of type C.

For any term, at least one type can be determined. Determination of the type(s) of a term t is based on the following:

- if t is a constant and $t < C$, then C is a type of t.
- if t is a typed term of type C, then C is the type of t.

Obviously, if t is a typed Skolem term of type C, then $t < C$.

The following notions are introduced here.

Definition 7 (Inclusion). A representation element x_l *includes* a representation element x_h if $G(x_l) \supseteq G(x_h)$.

Definition 8 (Overlap). A representation element x_l *overlaps* with a representation element x_h if $G(x_h) \not\subset G(x_l)$, $G(x_l) \not\subset G(x_h)$ and $G(x_l) \cap G(x_h) \neq \varnothing$.

Also, the following simple lemma is proved.

Lemma 2. $\{t_k\} \subseteq I(C_m)$ iff either (a) $t_k < C_m$ or (b) $\exists\ C_k$: $t_k < C_k$ and $C_k << C_m$.

Proof.

<= (a) If $t_k < C_m$, then $\{t_k\} \subseteq I(C_m)$. (b) If $t_k < C_m$ and $C_k << C_m$, then $\{t_k\} \subseteq I(C_k) \subseteq I(C_m)$ (Lemma 1, Section 2.3).

=> Since $\{t_k\} \subseteq I(C_m)$, t_k is a constant, that represents an instance. There are two cases. (a) $t_k < C_m$. (b) t_k is not an instance of C_m. Then, since t_k belongs to the C_m graph, there is a class C_k that belongs to the C_m graph such that $t_k < C_k$. Because C_k belongs to the C_m graph, $C_k << C_m$.

4.2.2 Definitions

In this subsection, we give definitions for the specialisation types between axioms provided by SILO. They deal with full MPL and only one (Def 12) is restricted to positive axioms that refer to only one attribute. They are based on an axiom-to-axiom model which is closer to the slot-to-slot model of frame-based systems.

Due to the large number of possible specialisation cases between axioms, it is impractical or even impossible to construct a system that is able to represent any possible case of any specialisation in a hardwired and implicit way. So, the definitions provided may not cover all possible cases. An effort to overcome this would lead either to introduction of several extra constructs and/or to very complicated, hence incomprehensible, definitions and expensive implementation. To avoid them, explicit user-based means are employed in SILO. This is the motivation behind the user-definable part of the inheritance mechanism (see Section 5).

In the following, by a_h we denote an axiom higher up and by a_l an axiom lower down in a hierarchy that belong to the context of the current object O_c. That is, a_l belongs to the current theory S_c and a_h is an axiom higher up considered for inheritance. Also, $H(a_i)$ represents the set of the self-literals of a_i. Finally, by p_s, p_m we represent a single-valued, a multi-valued attribute/predicate respectively.

Addition is the natural introduction of new knowledge in an object and needs no definition as well as no detection.

Extension concerns only multi-valued attributes/predicates. Various types of extension are defined as follows:

Definition 9 (Pure Extension). An axiom a_l is a *pure extension* of an axiom a_h if $a_h[p_m]^+$, $a_l[p_m]^+$ and $G(a_h) \cap G(a_l) = \varnothing$ (*positive extension*) or $a_h[p_m]^-$, $a_l[p_m]^-$ and $G(a_h) \cap G(a_l) = \varnothing$ (*negative extension*).

Definition 10 (Including Extension). An axiom a_l is a *including extension* of an axiom a_h, where $a_l \neq a_h$, if $a_h[p_m]^+$, $a_l[p_m]^+$ and a_l includes a_h (*positive extension*), or $a_h[p_m]^-$, $a_l[p_m]^-$ and a_l includes a_h (*negative extension*).

Definition 11 (Overlapping Extension). An axiom a_l is an *overlapping extension* of an axiom a_h if $a_h[p_m]^+$, $a_l[p_m]^+$ and a_l overlaps with a_h (*positive extension*), or $a_h[p_m]^-$, $a_l[p_m]^-$ and a_l overlaps with a_h (*negative extension*).

Substitution is defined as follows:

Definition 12 (Substitution). An axiom a_l is a *substitution* for an axiom a_h if $a_h[p_s]^+$, $a_l[p_s]^+$ and $a_h \neq a_l$.

Definition 12 is based on the fact that there cannot be more than one positive definition of a single-valued attribute/predicate in the context of an object. This is not the case for negative definitions. Also, Definition 12 concerns positive axioms that refer (Def 6a) to only one attribute. It is not easy to extend it to more that one attribute or to multi-valued attributes/predicates in an implicit way. These are substitution cases to be handled by the user-definable part of the inheritance mechanism.

Refinement is meaningful only for multi-valued attributes/predicates and is defined as follows:

Definition 13 (Refinement). An axiom a_l is a *refinement* of an axiom a_h, where $a_l \neq a_h$, if $a_h[p_m]^+$, $a_l[p_m]^+$ and a_h includes a_l (*positive refinement*) or if $a_h[p_m]^-$, $a_l[p_m]^-$ and a_h includes a_l (*negative refinement*).

In SILO, we interpret negation as a means of expressing exceptions, hence the notion of exception is related to that of logical inconsistency. So, S_c includes an exception to an axiom a_h if and only if $S_c \cup \{a_h\}$ is inconsistent, or in other words if $(\sim a_h)$ belongs to or can be proved from S_c, symbolically S_c Ã $(\sim a_h)$. However, given the exponential explosion of the resolution process and the semidecidability of logic, this may lead to prohibitively expensive computations. Therefore, we introduce some shortcuts based on an axiom-to-axiom exception model, which is closer to the slot-to-slot exception model of the standard object-based systems. These shortcuts are only

used for inheritance purposes. The reasoning process itself is based on the resolution refutation procedure [12].

We define the two types of *exception* as follows.

Definition 14 (Full Exception). An axiom a_l is a *full exception* to (or *fully inconsistent* with) an axiom a_h, if $\exists\ l_{l_i} \in H(a_l)$, $l_{h_i} \in H(a_h)$, $i=1..n$,: l_{l_i} includes $(\sim l_{h_i})$, $i=1..n$, and $(a_l - \{l_{l_1},.., l_{l_n}\})$ includes $(a_h - \{l_{h_1},.., l_{h_n}\})$.

In the case of unit axioms (i.e. slot-axioms) the above definition reduces to the following:

Definition 14a (Unit Full Exception). A unit axiom a_l is a *full exception* to (or *fully inconsistent* with) a unit axiom a_h if a_l includes $(\sim a_h)$.

In contrast to full exception, that concerns both single- and multi-valued attributes/predicates, partial exception concerns only multi-valued attributes/predicates:

Definition 15 (Partial Exception). An axiom a_l is a *partial exception* to (or *partially inconsistent* with) an axiom a_h, if $a_h[p_m]$, $a_l[p_m]$ and $\exists\ l_{l_i} \in H(a_l)$, $l_{h_i} \in H(a_h)$, $i=1..n$,: l_{l_i} overlaps with $(\sim l_{h_i})$, $i=1..n$, and $(a_l - \{l_{l_1},.., l_{l_n}\})$ overlaps with $(a_h - \{l_{h_1},.., l_{h_n}\})$.

In the case of unit axioms (i.e. slot-axioms) the above definition reduces to the following:

Definition 15a (Unit Partial Exception). A unit axiom a_l is a *partial exception* to (or *partially inconsistent* with) a unit axiom a_h, if $a_h[p_m]$, $a_l[p_m]$ and a_l overlaps with $(\sim a_h)$.

4.3 Conflicts and complete inheritance

4.3.1 Conflicting and redundant axioms

A conflict, from the point of view of complete inheritance, refers to all the consequences of an axiom. So, we give the following definition of a *conflict*:

Definition 16 (Conflict). An axiom a_h is *conflicting* with an axiom a_l if a_l is either a substitution for or a refinement of or a full exception to a_h.

Apart from the notion of conflicting axioms, we also introduce the notion of *redundant* axioms. Redundancy is mainly due to including extensions. Thus, we have the following definition:

Definition 17 (Redundancy). An axiom a_h is *redundant* if there is an axiom a_l that is either an including extension of or equal to a_h.

As far as attribute declarations are concerned, we consider that any attribute declaration lower down is conflicting with any attribute declaration higher up for the same attribute.

4.3.2 Detection theorems

From Definitions 16 and 17, it is clear that detection of conflicting and redundant axioms amounts to detection of the corresponding specialisations between axioms. Detection of substitution is straightforward from its definition. However, detection of the other specialisations reduces to the detection of the inclusion relation between axioms or literals of axioms. In this subsection, a few theorems that aim at the detection of the inclusion relation are presented.

First, we prove a theorem that is the basis for terms inclusion detection, which in turn is the basis for literals and that for axioms inclusion detection.

Theorem 1 (Terms Inclusion). A term t_l of type C_l includes a term t_h of type C_h iff (a) $C_h << C_l$ or (b) $t_h < C_l$ or (c) $t_h \equiv t_l$.

Proof.

<= Since t_l is a variable, $G(t_l) = I(C_l)$. In (a), $C_h << C_l \Rightarrow I(C_h) \subseteq I(C_l)$ (Lemma 1). If t_h is a variable, $G(t_h) = I(C_h)$, hence $G(t_h) \subseteq G(t_l)$. If t_h is not a variable, then $t_h < C_h$ which implies $G(t_h) = \{t_h\} \subseteq I(C_h)$ (Lemma 2), hence again $G(t_h) \subseteq G(t_l)$. In (b), $t_h < C_l \Rightarrow G(t_h) \subseteq I(C_l)$ (Lemma 2), hence $G(t_h) \subseteq G(t_l)$.

=> Since t_l includes t_h, $G(t_h) \subseteq G(t_l)$. There are two possible cases for each of t_l, t_h : to be or not to be a variable. (1) t_l is a variable. (1.1) t_h is also a variable; then $G(t_h) = I(C_h) \subseteq G(t_l) = I(C_l)$ which implies $C_h << C_l$ (Lemma 1). (1.2) t_h is not a variable; then $G(t_h) = \{t_h\} \subseteq G(t_l) = I(C_l)$ which implies either $t_h < C_l$ or $C_h << C_l$ (Lemma 2). (2) t_l is not a variable. (2.1) t_h is a variable; then $G(t_h) \subseteq G(t_l) = \{t_l\}$ which, given that $G(t_h) \neq \varnothing$, implies

$G(t_h) = I(C_h) = \{t_l\}$, which in turn implies $t_h < C_l$. (2.2) t_h is not a variable; then $G(t_h) = \{t_h\} \subseteq G(t_l) = \{t_l\}$ which implies $t_h \equiv t_l$.

The following two theorems concern literals and axioms inclusion. Their proofs are omitted, as obvious.

Theorem 2 (Literals Inclusion). A literal l_l includes a literal l_h iff they have the same sign, the same predicate/attribute and the value arguments of l_l include the corresponding value arguments of l_h .

Theorem 3 (Axioms Inclusion). An axiom a_l includes an axiom a_h iff the literals of a_l include the literals of a_h .

4.3.3 Attribute/predicate overriding

After detection of conflicting or redundant axioms, *attribute/predicate overriding* is employed in SILO, as in object-based systems [26, 19]. Overriding in SILO demands that an axiom a_l lower down overrides any conflicting or redundant axiom a_h stored higher up in the hierarchy. The same holds for attribute declarations.

4.3.4 Examples

In this subsection, we give a number of examples illustrating realisation and detection of various specialisations. The examples refer to Fig.2, where `human` is a subclass of `mammal` and `john`, `mike` are instances of `human`, and only necessary knowledge is presented.

So, ((son man)) and ((lives-in ?x ?self) (~ (works-in ?x ?self))) in `human` are additions of knowledge.

The slot-axiom ((likes ?x:leisure-thing john)) in `john` is a positive including extension to ((likes ?x:game ?self)) in `human` (Def. 10), because they are not equal, refer to the same multi-valued attribute 'likes' and the first includes the second, since "?self" is considered to be bound to "john" and is given that `leisure-thing` includes `game` (`game` << `leisure-thing`). So, ((likes ?x:game ?self)) is redundant (Def. 17). In contrast, ((eats ?x:meat ?self)) in `human` is a positive pure extension to ((eats ?x:vegetable ?self)) in `mammal` (Def. 9). In the later case there is no redundancy, because $G(a_h) \cap G(a_l) = \varnothing$, given that `vegetable` and `meat` are disjoint classes.

Furthermore, ((~ (trusts ?x:woman mike))) in `mike` is a negative overlapping extension to ((~ (trusts ?x:politician ?self))) in `human`, because they are negative axioms and the first overlaps with the second (Def. 11), given that `politician` and `woman` are not disjoint classes, that is they may have common instances.

mammal
((num-of-legs 4 ?self))
((likes swimming ?self))
((eats ?x:vegetable ?self))

human
attributes
((son man))
axioms
((num-of-legs 2 ?self))
((likes ?x:game ?self))
((eats ?x:meat ?self))
((~ (trusts ?x:politician ?self)))
((lives-in ?x ?self) (~ (works-in ?x ?self))
((plays ?x ?self) (~ (likes ?x:game ?self)))
((origin asia ?self) (origin africa ?self)
(~ (colour dark ?self)))

john
attributes
((son doctor) (1 1))
axioms
((likes ?x:leis-thing john))
((origin europe john))
((~ (eats ?x:anim-prod john)))
((~ (trusts ?x:parl-memb john)))
((plays ?x john)
(~ (likes ?x:table-game john)))

mike
attributes
((son man) (2 !))
axioms
((eats beef mike))
((~ (likes swimming mike)))
((~ (plays ?x mike)) (~ (likes ?x:leis-thing mike)))
((~ (trusts ?x:woman mike)))
((lives-in ?y mike) (~ (father ?x mike))
(~ (lives-in ?y:city ?x)))

Fig.2 Knowledge representation in SILO: Example 1

The slot-axiom ((num-of-legs 2 ?self)) in `human` is a substitution for ((num-of-legs 4 ?self)) in `mammal` (Def. 12), because they positively refer to the same single-valued attribute 'num-of-legs'. Also, the method-axioms ((lives-in ?y mike) (~ (father ?x mike)) (~ (lives-in ?y:city ?x))) in `mike` and ((origin europe john)) in `john` are substitutions for ((lives-in ?x ?self) (~ (works-in ?x:city ?self)) and ((origin asia ?self) (origin africa ?self) (~ (colour dark ?self))) in `human` respectively, because they positively refer to the same single-valued attribute 'lives-in' and 'origin' respectively.

Additionally, the slot-axiom ((eats beef mike)) in `mike` is a positive refinement of ((eats ?x:meat ?self)) in `human` (Def. 13), because they are not equal, positively refer to the same attribute and the second includes the first, since their literals have the same predicate, the same sign and is given that `meat` includes `beef` (`beef` < `meat`). Also, the method-axiom ((plays ?x john) (~ (likes ?x:table-game john)) in `john` is a positive refinement of ((plays ?x ?self) (~ (likes ?x:game ?self)) in `human`, because they are not equal, positively refer to the same attribute 'plays' and the second includes the first, as their first literals are equal ("?self" is considered to be bound to "john") and (~ (likes ?x:game ?self) includes (~ (likes ?x:table-game john) given that `game` includes `table-game` (`table-game` << `game`). On the other hand, (~ (trusts ?x:parliament-member john)) in `john` is a negative refinement of (~ (trusts ?x:politician ?self)) in `human`, because they are not equal, refer to the same attribute and the second includes the first, given that `politician` includes `parliament-member`.

The slot-axiom ((~ (likes swimming mike))) in `mike` is a unit full exception to ((likes swimming ?self)) in `mammal` (Def. 14a), because (~ (likes swimming mike)) is equal to (~ (likes swimming ?self)) ("?self" is considered to be bound to "mike"). Also, ((~ (eats ?x:animal-product john))) in `john` is a unit full exception to ((eats ?x:meat ?self)) in `human`, because the first includes the negation of the second, given that `animal-product` includes `meat`. Also, ((~ (plays ?x mike)) (~ (likes ?x:leisure-thing mike))) in `mike` is a full exception to ((plays ?x ?self) (~ (likes ?x:game ?self))) in `human` (Def. 14), given that `leisure-thing` includes `game`, because (~ (plays ?x mike)) is equal to (~ (plays ?x ?self)) and (~ (likes ?x:leisure-thing mike)) includes (~ (likes ?x:game ?self)).

Finally, the attribute declarations ((son doctor) (1 1)) in `john` and ((son man) (2 !)) in `mike` are conflicting with ((son man)) in `human`, because they refer to the same attribute/predicate.

4.4 Atomic inheritance and consequence retraction

Overriding as well as the previously discussed conflict detection techniques concern complete inheritance. They are not appropriate to deal with cases where atomic inheritance should be considered, that is with cases where not the axioms themselves, but some of their atomic consequences should be only inherited. Such cases are those concerning partial exceptions/inconsistencies (e.g. like the ones described by Def. 15).

It is easy to see from Definitions 14 and 15 that if a_l is a partial exception to a_h, it is not an exception to a_h. So, a_l and a_h are not conflicting, according to Def. 16, although they have conflicting ground instantiations, and they are both inherited. This, however, may lead to inconsistent answers (atomic consequences). Also, it is easy to see that overriding cannot solve the problem without loosing knowledge and reducing specialisation flexibility.

We distinguish two categories of partial inconsistencies. The first category includes *self-context* partial inconsistencies, that is cases where the involved axioms consist of only self-literals. For example, in the partial hierarchy of Fig.3 (where `dad-mimic` is a subclass of `man`, `paul` is an instance of `man` and `peter`, `jacob` instances of `dad-mimic`), ((~ (trusts ?x:politician paul))) in `paul` is partially inconsistent with ((trusts ?x:man ?self)) in `man`, since `politician` and `man` are not disjoint, that they may have common instances (e.g. `m3` in Fig.1). This means that they have conflicting ground instantiations. Therefore, the answer to both queries (trusts m3 paul) and (~ (trusts m3 paul)) will be T (true).

The second category includes *inter-context* partial inconsistencies, that is cases where either of the involved axioms has at least one message literal. In these cases, in contrast to self-context ones, the atomic consequences are due to knowledge (axioms) belonging to context(s) of object(s) other from the current which is(are) accessed only via message passing. For example, the method-axiom ((plays ?y ?self) (~ (father ?x ?self)) (~ (plays ?y ?x))) in `dad-mimic` (Fig.3) is inherited by `peter`. Thus, we get four answers to the query (plays ?x peter), namely (plays football peter), (plays piano peter), (plays flute peter) and (plays chess peter), due to message passing of the (message) literal (plays ?y ?x) to `paul` (since "?x" is eventually bound to "paul" after

resolution of (~ (father ?x ?self)))). However, not all of those atomic consequences (solutions) are acceptable, since Peter does not actually play everything that his father plays, as it is denoted by the negative axioms stored in `peter`.

man
((sex male ?self))
((trusts ?x:man ?self))

dad-mimic
((plays ?y ?self)
(~ (father ?x ?self))
(~ (plays ?y ?x)))

paul
((plays football paul))
((plays piano paul))
((plays flute paul))
((plays chess paul))
((~ (trusts ?x:politician paul)))

peter
((father paul peter))
((~ (plays football peter)))
((~ (plays ?x:wind-instr peter)))

jacob
((plays ?x jacob)
(~ (plays ?x peter)))
((~ (plays chess jacob))

Fig.3 Knowledge representation in SILO: Example 2

To remedy the above deficiencies, since axiom overriding is not adequate, another inheritance technique, called *consequence retraction* is introduced in SILO. Consequence retraction is very similar to 'solution invalidation', introduced in [21]. We could say that consequence retraction is a generalisation of solution invalidation.

Definition 18 (Consequence Retraction) An atomic consequence φ_i found during a reasoning process in the context of an object O_c is retracted if $S = S_c \cup \{\varphi_i\}$ is inconsistent, that is $(\sim\varphi_i)$ belongs to S_c or S_c Ã $(\sim\varphi_i)$.

This means that an atomic consequence is retracted if it successfully resolves in S_c, that is the empty clause is produced. When a message is sent from the current object O_c (sender) to an object O_r (receiver) and an atomic consequence φ_i is derived within the context of O_r, then φ_i is retracted if it successfully resolves in S_c.

Using consequence retraction in the above example, the answer to query ((trusts m3 paul)) is F (false), because it resolves with ((~ (trusts ?x:politician paul))) and the empty clause is produced. Also, the answers to query (plays ?x peter) are only (plays

piano peter) and (plays chess peter), because the other two candidate answers are retracted. More specifically, (plays football peter) is retracted as successfully resolving with (~ (plays football peter)), and (plays flute peter) as successfully resolving with (~ (plays ?x:wind-instrument peter)), given that `flute` < `wind-instrument`. Because of the same consequences retraction within `peter` plus retraction of (plays chess jacob) within `jacob`, the answer to the query (plays ?x jacob) is only (plays piano jacob). Notice that not only solutions, but also intermediate atomic consequences are retracted, in contrast to solution invalidation. Additionally, while consequence retraction naturally exploits negation in its implementation, solution invalidation uses extra objects (units) instead [21].

Also, notice that the above axioms create a partial exception situation that cannot be detected based on Definition 15. This is because it is not able to know in advance whether what Peter's father plays includes the ones excepted in `peter`. So, consequence retraction can handle cases of partial exception that cannot be literally detected. We can say that consequence retraction is actually a way of implementing atomic exceptions and thus can cover even cases of full exceptions that cannot be detected based on the definitions. This makes consequence retraction a very powerful and general technique.

4.5 Exception by negation and state change

As it is clear so far, negation is used as the means for representing exceptions in SILO. Thus, any negative axiom (clause) represents either a full exception to another axiom (Def. 14), which is then overridden, or a partial exception to another axiom (Def. 15), which is not overridden but its inconsistent atomic consequences are retracted to restore consistency. We call this representation scheme, introduced here, *exception by negation*.

Exception by negation facilitates representation of state changes in problems like those relating to planning in the blocks world (see e.g. [9 Ch.11]). The state change in the example of Fig.4 can be represented as in Fig.5, where `state2` is a subclass of `state1`.

The same example is reported in [21] and MULTILOG [18], where 'solution invalidation' and 'inheritance with exceptions' respectively are employed. In SILO, the multi-valued predicates 'has-on-table', 'has-on' and 'has-free' are used, instead of the standard 'on-table', 'on' and 'free', for better readability. The system in [21] uses extra units (objects), where invalid solutions (facts) are stored. In this way, however, change state is dynamically/indirectly represented via solution invalidation. That is, all of the facts in `state1` are inherited by `state2` and only during reasoning exceptions are activated. MULTILOG provides a direct representation, where axioms to be excepted are specified in the specialisation/inheritance relation defined between the two objects. However, in that case, facts that were changed are not explicitly present in the objects, as in SILO. In SILO, a history of the changes is kept in the objects.

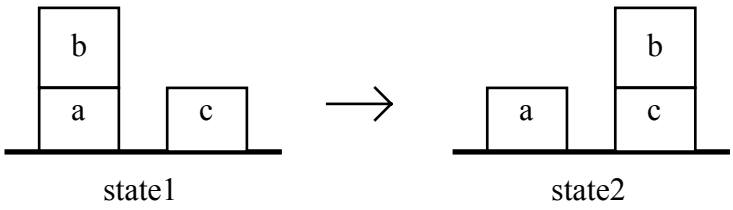

Fig.4 A state change in a blocks world.

state1
((has-on-table a ?self))
((has-on b a ?self))
((has-free b ?self))
((has-on-table c ?self))
((has-free c ?self))

state2
((has-free a ?self))
((has-on b c ?self))
((~ (has-free c ?self)))
((~ (has-on b a ?self)))

Fig.5 Representation of the state change of Fig.4 in SILO

So, exception by negation can be seen as providing a solution to the 'frame problem' (see [9 Ch.11]). Each new state can be represented by creating a new object containing

the new facts together with the negations of the invalid old facts and inheriting from the previous state all the facts but those excepted because their negations are present. Consequently, evolution of a state by successive changes can be represented as a branch of classes in a hierarchy.

4.6 Inheritance order

If the conflicting axioms are not within classes that belong to the same path in a hierarchy, but within classes/superclasses of an instance/class that belong to different paths, the situation is more complicated. In those cases inheritance order should be also considered.

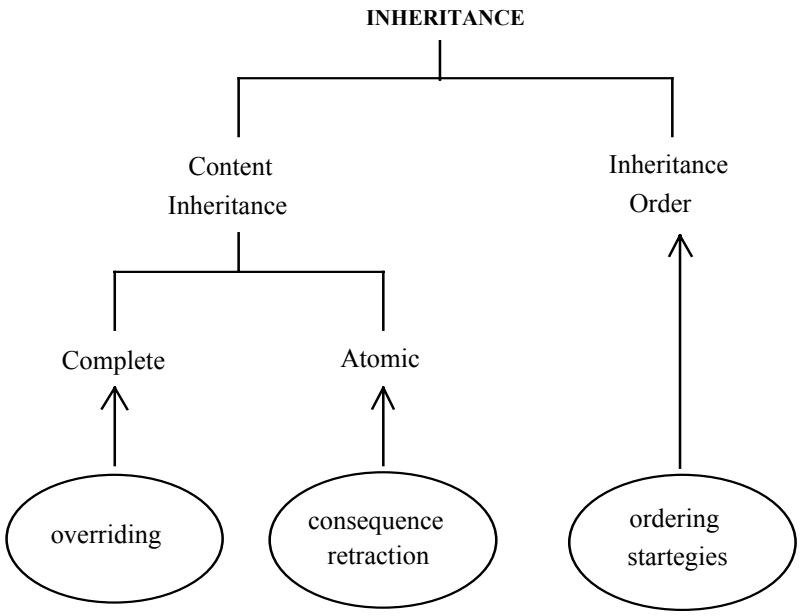

Fig. 6 Inheritance Aspects and Techniques

The order in which an instance/subclass inherits from its classes/superclasses is very important, as the first occurrence of an axiom overrides all subsequent conflicting occurrences higher up. In this case, an ordering strategy is required to define the precedence list of the classes/superclasses of an object. The *precedence list*

$\downarrow_{O_i}$ of an object O_i is an ordered set of the superobjects of the object that determines the *inheritance path* to be followed. A superobject of an object O_i is any object higher up in the hierarchy that belongs to a path from the root to O_i. A breadth-first left-to-right strategy is used as the default strategy (see next section), to determine the inheritance path. For example, the default precedence list of `m3` in the hierarchy of Fig.1 is $\downarrow_{m_3}$ = {`man, politician, human, mammal, animal, object`}.

The inheritance aspects and techniques discussed in Section 4 are depicted in Fig.6. Cases that cannot be handled by the above techniques are left to the user-definable component of the inheritance mechanism, discussed in the next section.

5. Inheritance Control Representation

5.1 The need for explicit control

Overriding is used as a general technique for resolving conflicts in SILO. However, it has some deficiencies. First, it is too difficult to devise general definitions so that be able to detect all cases of conflicting axioms. Although consequence retraction, the other general technique, can solve part of the problem by declaring exceptions to atomic consequences of the axioms, it cannot cover all cases in a natural and/or efficient way. Also, it cannot handle all types of specialisation.

On the other hand, selection of the inheritance path in cases of multiple inheritance is not an easy problem. There are several strategies that propose different ways for solving it [19]. Some of them are general search methods, like depth-first and breadth-first, which, however, are not adequate. For example, for the cases (a) and (b) in Fig.7, a breadth-first left-to-right strategy gives the inheritance paths A B C D E and A B C E D respectively. Only the first is acceptable, since in the second A inherits from E before it inherits from D (a subclass of E). Also, a depth-first strategy gives the lists A B D E C and A B E C D. None of them is acceptable, for similar reasons.

There are other methods that result in a linearisation of the hierarchy of an object, based on a depth-first strategy and some general heuristics like preserving modularity and local multiplicity [5, 19] that solve the above problems. However, there are still cases that cannot be satisfied, like case (c) in Fig. 7. In that case, where we want A to

inherit first from B and then from C, B to inherit first from D and then from E, and C to inherit first from E and then from D, those methods fail to produce a path. From the two possible paths A B C D E and A B C E D none is satisfactory. The first is not satisfactory when there is conflicting information in C, D and E, while the second when there is conflicting information in B, D and E.

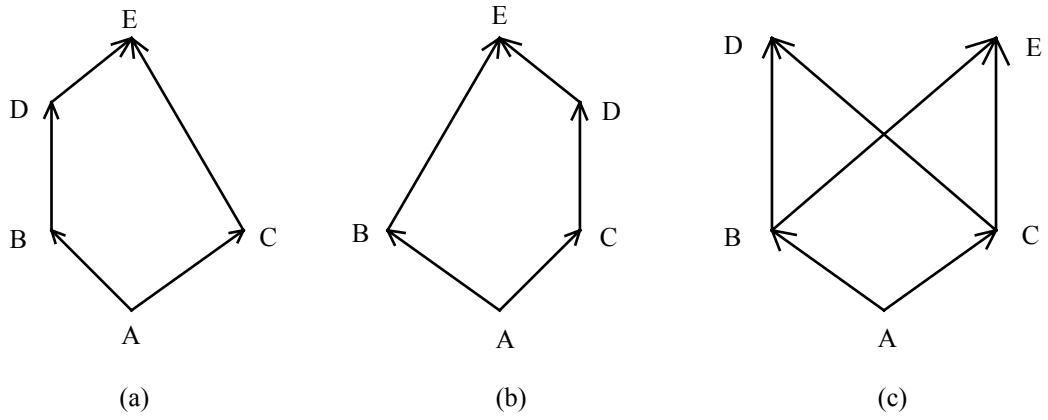

Fig.7 Interesting cases of multiple inheritance

This is mainly due to local domain-dependent irregularities that cannot be captured by general strategies. Thus, it is unlikely that a single hardwired strategy will be satisfactory in all cases, both in semantics and efficiency. For the above reasons, local, explicit and separate representation of control-knowledge is employed in SILO.

5.2 The meta-level model

The advantages of explicit and separate representation of control knowledge have been pointed out by a number of researchers, e.g. [3, 16]. The most well-known and advantageous architecture for implementing such a separation is that offered by *meta-level systems* [28]. This type of architecture provides a separate object-level and meta-level interpreter. The object-level interpreter reasons about the domain knowledge, whereas the meta-level interpreter reasons about how to use the domain knowledge. A main problem with meta-level systems is the meta-level overhead: the increase in

computing cost per object-level step, due to the corresponding meta-level steps, often exceeds the computational gain due to the reduction of the number of the object-level steps [28].

A kind of a meta-level architecture based on a *partial reflection* between object-level and meta-level is adopted in SILO. This approach suggests a partial reflection via a set of programmable steps in the object-level computational cycle. At certain steps in the object-level cycle, the system reflects at the meta-level to make decisions about the inference strategy used at the object-level. This kind of architecture achieves a satisfactory balance between flexibility and efficiency [28].

The object-level language of SILO is the integrated language used for the description of the structure-part and the knowledge-part of an object, described in Section 3. Its meta-level language, used for description of the control-part of an object, consists of a number of functions, called *meta-functions*, which determine various components of the overall control regime, concerning inheritance and logical deduction. Thus, we distinguish between *inheritance control meta-functions* and *deduction control meta-functions* that implement programmable steps in the computational cycle[3]. A prototype of a SILO's kernel that has been implemented in CommonLisp [25] provides a number of user-definable Lisp functions for inheritance and deduction control.

5.3 Inheritance control meta-functions

Determination of the meta-functions for inheritance control is based on a simple abstract analysis of inheritance control (Fig.8). Two control aspects of inheritance are distinguished that correspond to the two aspects of inheritance, *content inheritance* and *inheritance order*. This suggests that the system should provide programmable steps (meta-functions) for controlling both inheritance aspects.

Furthermore, the system should provide the user the capability of defining both *global* and *local* control regimes. Global control refers to inheritance rules to be

[3]The same idea is used in ACT-P [13], where it is treated in more detail. SILO's deduction control is based on that introduced in [13].

applied to all objects, whereas local control to rules to be applied to a specific object and its (super)classes. Thus, global regimes cannot take into account local irregularities, whereas local can. So, as it is clear from Fig.8 (bottom level), meta-functions are required for the four terminal nodes of the analysis tree. One meta-function for each node is employed.

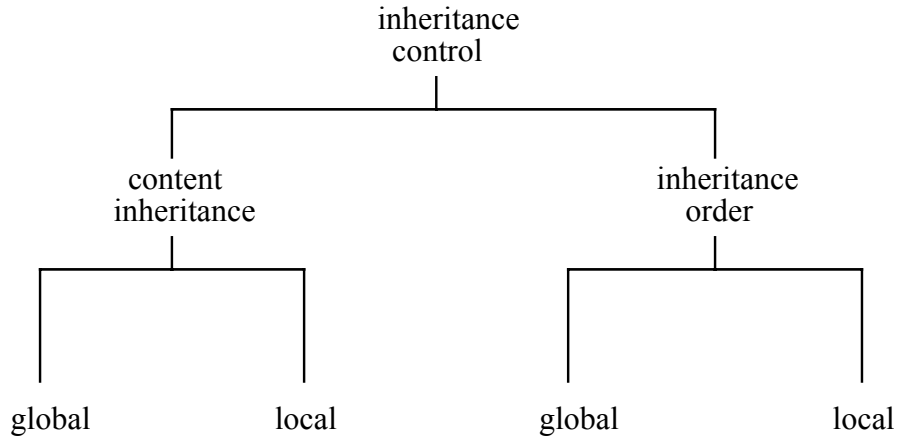

Fig.8 Inheritance Control Analysis

The meta-functions used in SILO for handling content inheritance are: **l-inherit-axioms**, for local control of inheritance of axioms, and **g-inherit-axioms**, for global control of inheritance of axioms. While **g-inherit-axioms** is used to specify domain-independent policies, **l-inherit-axioms** is used to specify domain-specific ones. For handling inheritance order, SILO provides the meta-functions **l-order-classes** and **g-order-classes**, for local and global control respectively. By redefining these meta-functions the user can define his/her own inheritance rules either locally or globally. Definitions of local activity meta-functions are stored in the control-part of the objects. Global activity meta-functions are globally defined. The inheritance control meta-functions are briefly described in the Appendix.

To facilitate definition of the meta-functions, special built-in primitives, called *meta-primitives*, are provided that can access object-internal information. A few examples of meta-primitives are provided in Section 5.5.

5.4 Inheritance and reasoning

A reasoning process starts off when a message (theorem) is sent to an object O_c , which becomes the current object. Inheritance is closely related to the reasoning process. It is the means of extending the local theory of the current object when it fails to prove a theorem (query), while preserving consistency. To preserve consistency, and in general to resolve conflicts, not all of the axioms higher up are inherited. This results in nonmonotonic extensions of a local theory and facilitates *nonmonotonic reasoning*, which thus is very naturally performed in SILO.

The inheritance principles for detecting conflicting or redundant axioms, specified in Section 4, are implemented as a number of *inheritance rules*. We represent by $\vdash_b(S_i)$ and $\vdash_u(S_i)$ the action of the *built-in* and the *user-defined* inheritance rules on S_i respectively. Each of $\vdash_b$, $\vdash_u$ takes as input a theory set and gives as output the axioms that should be inherited, that is pass the filter of the corresponding inheritance rules. Apart from the current theory set S_c, two auxiliary sets, S_p and S_r, representing the passed and rejected axioms respectively of the *next object* C_n in the precedence list $\downharpoonleft_{O_c}$ are used. The proof procedure is briefly described below.

(1) Determine the precedence list $\downharpoonleft_{O_c}$.
(2) Set $S_c = \llcorner_{O_c}$; start a proof process.
(3) If the theorem is proved from S_c, then check consequence retraction in S_c and stop (success).
(4) If $\downharpoonleft_{O_c}$ is empty, stop (failure).
(5) Remove the first element of $\downharpoonleft_{O_c}$ and specify C_n.
(6) Make $S_p = \vdash_b(\llcorner_{C_n})$ and $S_r = \llcorner_{C_n} - \vdash_b(\llcorner_{C_n})$.
(7) Make $S_p = \vdash_u(S_p) \cup \vdash_u(S_r)$ and $S_r = (S_r - \vdash_u(S_r)) \cup (S_p - \vdash_u(S_p))$[4].
(8) Make $S_c = S_c \cup S_p$; set S_p , $S_r = \varnothing$; start a new proof process.
(9) Go to step 3.

The steps which the meta-functions are involved in are indicated in the Appendix. The precedence list $\llcorner_{O_c}$ is computed by the **g-order-classes** function according to local arrangements suggested by the **l-order-classes** functions stored within the superobjects

[4]All occurrences of S_p and S_r in the right-hand side of both assignments are supposed to represent their values in (6).

of O_c. The meta-primitive **get-ordered-sups** is fundamental to this purpose. It takes as arguments an object's name and the theorem to be proved, and returns the superclasses of the object in an order specified by the user-defined local control meta-function **l-order-classes**. As said, the default definitions of **g-order-classes** and **l-order-classes** suggest a breadth-first left-to-right strategy.

The user-defined meta-functions for content inheritance are applied in the order: **g-inherit-axioms** , **l-inherit-axioms**. The arguments of these meta-functions are such that decisions based on the hardwired rules can be retracted, as it is also clear from the above described proof procedure. Also, because the query (theorem) to be answered (proved) is provided as an argument to all of the above functions, problem-specific rules can be also defined.

5.5 Examples

5.5.1 Breadth-first ordering

To illustrate inheritance control in SILO, some examples are presented. As a first example, implementation of the default inheritance order strategy of SILO is given. This is done by defining the meta-functions **l-order-classes** and **g-order-classes** as follows:

```
; Local control.
  ; sups represents the classes of O_c in the order specified by the user at creation time.
  ; theorem represents the theorem (query) to be proved (answered).

(defun l-order-classes (sups theorem)
   sups)

; Global control.
; Top level function
  ; sups represents the classes of O_c in the order specified by l-order-classes.
  ; p-list represents ↓O_c.

(defun g-order-classes (sups theorem)
  (let ((p-list sups))
   (if (member 'object p-list) nil
      (let ((next-sups (order-next-sups p-list theorem)))
       (append p-list (g-order-classes next-sups theorem))))))
```

```
; Second level function

(defun order-next-sups (p-list theorem)
  (let ((next-sups nil))
    (dolist (sup p-list next-sups)
      (let ((ord-next-sups (get-ordered-sups sup theorem)))
        (setf next-sups (appendnew next-sups ord-next-sups))))))


  ; Auxiliary function for appending two lists, where duplicates are eliminated.

(defun appendnew (lista listb)
    (let ((new-listb nil))
      (dolist (elem listb (append lista (reverse new-listb)))
        (if (not (member elem lista)) (push elem new-listb)))))
```

5.5.2 The 'Nixon Diamond' problem

As a second example, we present how SILO can implement various strategies related to the well-known problem of 'Nixon Diamond' (see e.g. [19 Ch.7, 26]). The partial hierarchy and knowledge of Fig.9[5] are considered, where `nixon` is an instance of both `republican` and `quaker`, and 'behaviour' and 'political-side' are declared as multi-valued attributes. The question is what is Nixon's behaviour: (behaviour ?x nixon).

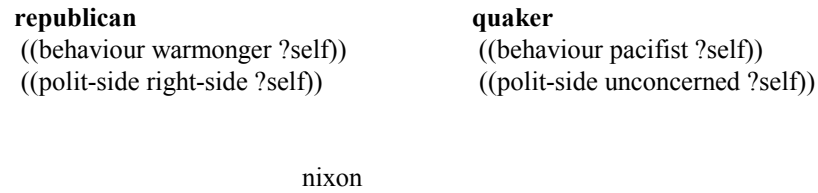

Fig.9 The Nixon Diamond Problem

In general, there are two approaches to the problem. According to the skeptical approach no decision is taken [27]. The question of determining whether Nixon is a pacifist or a warmonger remains unanswered. This can be easily resulted in SILO by not allowing either of the slot-axioms to be inherited by `nixon`, by means of local control. To this end, the **l-inherit-slots** is defined, and stored in `nixon`, as follows:

---

[5] Notice that there are other (non common) instances of `republican` and `quaker`, not depicted in the figure for the sake of simplicity.

; n-obj represents C_n, and p-axioms and r-axioms represent S_p and S_r respectively.

```
(l-inherit-axioms (n-obj p-axioms r-axioms theorem)
    (remove-if #'(lambda (axiom)
                         (and (slot-axiom axiom)
                              (eq (get-pred axiom) 'behaviour)))
                 p-axioms)),
```

where **slot-axiom** is a meta-primitive that checks if an axiom is a slot-axiom, and **get-pred** is a meta-primitive that gets the predicate of a unit axiom.

According to the second approach, the credulous approach, both answers are employed [27]. This can also be easily implemented in SILO. We simply allow for both axioms to be inherited. This is the default behaviour of SILO.

However, SILO can go further. Suppose that we want to express the fact that Nixon follows his quakerism as far as his 'behaviour' is concerned, but he follows his republicanism as far as 'political-side', another of his attributes, is concerned. This cannot be satisfied by a general strategy as the first requires the order of the classes of `nixon` to be {`quaker`, `republican`} and the second the reverse. SILO can easily represent this by defining and storing in `nixon` the meta-function **l-inherit-axioms** as follows:

```
(l-inherit-axioms (n-obj p-axioms r-axioms theorem)
  (cond ((eq n-obj 'republican)
         (remove-if #'(lambda (axiom)
                              (and (slot-axiom axiom)
                                   (eq (get-pred axiom) 'behaviour)))
                    p-axioms))
        ((eq n-obj 'quaker)
         (remove-if #'(lambda (axiom)
                              (and (slot-axiom axiom)
                                   (eq (get-pred axiom) 'political-side)))
                    p-axioms))
        (t p-axioms))))
```

None of the existing similar systems can offer a solution to this case of the problem.

6. Related Work and Discussion

There are a large number of systems that in some way combine logic and objects and use the notion of inheritance. Most of them are based on Horn-type logic (being

extensions of Prolog-like logic programming), do not use negation, and do not deal with conflicting information (all answers are acceptable), that is they adopt unrestricted non-determinism. Thus, their objects are unstructured, that is flat sets of logical expressions. Furthermore, they approach the combination of logic and objects from a programming point of view rather than that of knowledge representation, in contrast to SILO. Thus, their objects model is mostly based on the set theory rather than the prototype theory [19], hence it is not adequate for knowledge representation. For the purposes of this paper, we distinguish those systems in two broad categories.

The systems of the first category organise their objects (:sets of clauses) in a graph, where objects are connected via explicit inheritance relations. They do not necessarily impose a hierarchical structure on them and do not usually distinguish between classes and instances. For example, MULTILOG [18] organises its objects (worlds) via three inheritance relations: full inheritance, inheritance with exceptions, and default inheritance. Multiple inheritance is allowed. MULTILOG, however, does not use negation and has no mechanism to deal with atomic inheritance. Also, one cannot define a variety of combinations of inheritance (specialisation) relations (types) between objects (axioms) as in SILO. The system in [21] is based on the concept of a context as an ordered set of theories. A context is dynamically constructed and can be seen as a branch in a hierarchy. Predicate overriding and predicate extension are provided, by using explicit means. Also, solution invalidation is introduced, however it is achieved via extra objects, called constraints objects. This is done more generally, naturally and efficiently in SILO through consequence retraction and exception by negation. Moreover, because SILO uses typed terms, a group of atomic forms can be represented via a single axiom and also a group of solutions can be invalidated via a single axiom. Finally, multiple inheritance is not treated in [21]. In [20], a graph of objects is constructed using only two types of inheritance: full inheritance and overriding inheritance. Multiple inheritance is supported. However, it only deals with atomic inheritance. Although, default reasoning can be achieved, exception of axioms cannot be implemented.

The second category includes systems that, like SILO, create a hierarchical structure and usually distinguish between classes and instances. POL [8] realises different inheritance rules by using explicit declarations for different types of methods, such as normal, default and deterministic methods. Multiple inheritance is also supported, but certain specialisation types like knowledge refinement cannot be implemented. SPOOL [7] has a Flavors-like inheritance mechanism that provides facilities for method combination, but it is quite inflexible in representing knowledge. The LAP system [15] uses a fixed depth-first search strategy to order multiple parents of an object prior to overriding. In CPU [22], inheritance is explicitly expressed as meta-level knowledge via meta-objects (meta-units). Although this gives a great flexibility, it creates a high meta-level overhead. Finally, Plog [17] supports full and overriding inheritance only for classes. Also, multiple inheritance is provided only for classes.

7. Conclusions

In this paper, the inheritance mechanism of SILO is presented; it is an extension of an earlier work [14]. The mechanism consists of two components, a hardwired and a user-definable. The hardwired component comprises a number of inheritance rules dealing with complete inheritance. These rules filter out conflicting or redundant knowledge. Use of a kind of many-sorted logic for domain knowledge representation within objects greatly facilitates this task, so that simple, implicit and natural rules are produced. Axiom overriding is used for this purpose. Moreover, consequence retraction is used to handle atomic inheritance. Finally, a fixed ordering strategy is provided to determine the inheritance path.

The user-definable component comprises a number of user-definable functions, the meta-functions, which deal with complete inheritance and inheritance order. The user, by locally or globally (re)defining the meta-functions, can implement a variety of inheritance relations and ordering strategies. In this way, not only domain-specific, but also problem-specific irregularities can be represented. This gives SILO a great flexibility in representing specialisations in a hierarchy.

Also, since determination of the control components (meta-functions) is based on an abstract analysis, they are fully programmable and can implement all fundamental types of specialisation, it is guaranteed that SILO is adequate as far as inheritance is concerned.

A weak point of SILO's inheritance mechanism is the implicit and procedural nature of the inheritance rules, that may create some difficulties in realising their use. Also, the provided meta-functions are quite a few and, although they give great flexibility, they give no sufficient guidance to the user, mainly as far as inheritance order is concerned. This is actually the cost paid for its great flexibility. Moreover, the created code for control knowledge representation may be almost unreadable, due to the procedural nature of the meta-language, in contrast to a declarative one. A library of possible strategies would be an improvement to this point.

What SILO does not provide in its inheritance mechanism is the capability of implementing what is called 'method combination' in the object-oriented parlance [19, 26]. Also, it does not offer any specific constructs for representation of 'part-of' relations. These are directions for further research on and development of SILO's inheritance mechanism.

Appendix

Inheritance Control Meta-functions

control aspect	meta-function	arguments	output	step involved
content inheritance	l-inherit-axioms	next-object theorem passed-axioms rejected-axioms	updated passed-axioms	7
	g-inherit-axioms	next-object theorem passed-axioms rejected-axioms current-theory	updated passed-axioms	7
inheritance order	l-order-classes	superclasses theorem	ordered superclasses	1
	g-order-classes	superclasses theorem	precedence list	1

References

[1] L. Aiello, C. Cecchi and D. Sartini, Representation and use of metaknowledge, *Proceedings of the IEEE* 74 (1986) 1304-1321.

[2] D. G. Bobrow and M. Stefik, The LOOPS Manual: a data and object oriented programming system for Interlisp, Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13, Xerox PARC, Palo Alto, California (1983).

[3] W. Clancey, The advantages of abstract control knowledge in expert system design, *Proc. of the 3rd Annual Meeting of the AAAI* (1983) 74-78.

[4] A.G. Cohn, Taxonomic reasoning with many-sorted logics, *AI Review* 3 (1989) 89-128.

[5] R. Ducournau and M. Habib, Masking and conflicts, or to inherit is not to own, in: M. Lenzerini, D. Nardi and M. Simi, eds., *Inheritance Hierarchies in Knowledge Representation and Programming Languages* (John Wiley & Sons, 1991) 223-244.

[6] R. Fikes and T. Kehler, The role of frame-based representation in reasoning, *CACM* 28(9) (1985) 904-920.

[7] K. Fukunaga and S. Hirose, An experience with a Prolog-based object-oriented language, *Proc. of the OOPSLA'86*, as *SIGPLAN Notices* 21 (1986) 224-231.

[8] H. Gallaire, Merging objects and logic programming: relational semantics, *Proc. of the AAAI'86* (1986) 754-758.

[9] M. R. Genesereth and N. J. Nilsson, *Logical foundations of Artificial Intelligence* (Morgan Kaufmann, 1987).

[10] I. Hatzilygeroudis, IJCAI-91 workshop on objects and artificial intelligence, *AI Magazine* 15(2) Summer 1994 (1994) 86-87.

[11] I. Hatzilygeroudis, Knowledge representation and reasoning in a system integrating logic in objects, *Proc. of the 5th IEEE ICTAI* (1993) 160-167.

[12] I. Hatzilygeroudis, SILO: Integrating logic in objects for knowledge representation and reasoning, *International Journal on AI Tools* 5(4) (1996), forthcoming.

[13] I. Hatzilygeroudis and H. Reichgelt, ACT-P: a configurable theorem-prover, *Data & Knowledge Engineering* 12 (1994) 277-296.

[14] I. Hatzilygeroudis and H. Reichgelt, The inheritance mechanism of a system integrating logic in objects, *Proc. of the 6th IEEE ICTAI* (1994) 724-727.

[15] H. Iline and H. Kanoui, Extending logic programming to object programming: the system LAP, *Proc. of the 10th IJCAI* 1 (1987) 34-39.

[16] P. Jackson, H. Reichgelt and F. van Harmelen, eds., *Logic-based Knowledge Representation* (MIT Press, 1989).

[17] M. Jenkins and D. Chester, A combined object-oriented and logic programming tool for AI, *Proc. of the 5th IEEE ICTAI* (1993) 152-159.

[18] H. Kauffmann and A. Grumbach, MULTILOG: MULTIple worlds in LOGic programming, *Proc. of the 7th ECAI* 1 (1986) 291-305.

[19] G. Masini, A. Napoli, D. Colnet, D. Leonard and K. Tombre, *Object Oriented Languages*, The APIC Series (Academic Press, 1991).

[20] F.G. McCabe, *Logic and Objects* (Prentice Hall, 1992).

[21] P. Mello, Inheritance as combination of Horn clause theories, in: M. Lenzerini, D. Nardi and M. Simi, eds., *Inheritance Hierarchies in Knowledge Representation and Programming Languages* (John Wiley & Sons, 1991).

[22] P. Mello and A. Natali, Objects as communication Prolog units, *Proc. of the ECOOP'87*, as LNCS 276 (Springer Verlag, 1987) 181-191.

[23] M. Piff, *Discrete Mathematics: An introduction to software engineering* (Cambridge University Press, 1991).

[24] H. Reichgelt, *Knowledge representation: an AI perspective* (Ablex, 1991).

[25] Guy L. Steele Jr, *CommonLISP: The Language* (Digital Press, 1984).

[26] M. Stefik and D.G. Bobrow, Object-oriented programming: themes and variations, *AI Magazine* 7(4) Winter 1986 (1986) 40-62.

[27] D.S. Touretzky, J.F. Horty and R.H. Thomason, A clash of intuitions: the current state of nonmonotonic multiple inheritance systems, *Proc. of the 10th IJCAI* (1987) 476-482.

[28] F. van Harmelen, *Meta-level Inference Systems* (Pitman, 1991).