Published in the International Journal of AI Tools 5(4) (1996) 403-446.

SILO: INTEGRATING LOGIC IN OBJECTS FOR KNOWLEDGE REPRESENTATION AND REASONING

IOANNIS HATZILYGEROUDIS
University of Patras, School of Engineering, Dept of Computer Engineering & Informatics
26500 Patras, Greece
&
Computer Technology Institute (CTI), P.O. Box 1122, 26110 Patras, Greece

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

ABSTRACT
There have been a large number of systems that integrate logic and objects (frames or classes) for knowledge representation and reasoning. Most of those systems give pre-eminence to logic and their objects lack the structure of frames. These choices imply a number of disadvantages, as the inability to represent exceptions and perform default reasoning, and the reduction in the naturalness of representation. In this paper, aspects of knowledge representation and reasoning in SILO, a system integrating logic in objects, are presented. SILO gives pre-eminence to objects. A SILO object comprises elements from both frames and classes. A kind of many-sorted logic is used to express object internal knowledge. Message passing, alongside inheritance, plays a significant role in the reasoning process. Control knowledge, concerning both deduction and inheritance, is separately and explicitly represented via definitions of certain functions, called meta-functions.

Keywords: logic, frames, classes, knowledge representation, hybrid system, meta-level system.

1. Introduction

Logic-based knowledge representation formalisms, such as first-order logics, are extensively used for knowledge representation. One of their major advantage is their really great expressive power: logic can represent incomplete knowledge, such as negative and disjunctive knowledge [1, 2 Ch.3]. This makes logic very attractive in building knowledge bases. However, although this is true of small knowledge bases, it is not of larger ones. Knowledge bases written in a logic are just flat sets of propositions, with no structure, while larger knowledge bases require a structured way of representing knowledge. Since this feature is not offered by plain logics, developing and maintaining a relatively large logical knowledge base becomes a difficult task. Also, due to their increased expressive power, logic-based representation languages often have efficiency problems.

On the other hand, object-based formalisms, such as frame-based and class-based representations, have been successfully used in constructing even large knowledge bases. Their main advantage, compared to logic-based formalisms, is that they offer a structured way of representing knowledge. Furthermore, the structure of an object-based knowledge base corresponds to the structure of the represented real world [2 Ch.6]. This makes development and maintenance of knowledge bases easier and faster. Finally, object-based languages offer quite efficient implementations.

So, it seems that logic-based and object-based representations are, in some sense, complementary; logic has its drawbacks as a knowledge representation formalism in exactly those aspects where object-based representations are strong, and vice versa. Given this complementarity, it seems promising to combine logic and objects into a single system.

There have been a large number of attempts at combining logic and objects (see Section 9 for an account). Most of the efforts give pre-eminence to a logical framework, that is notions from object-based representations are somehow incorporated within or expressed via logic, and consider the combination from the programming point of view rather than that of knowledge representation. This, however, may lead to any of the following shortcomings: a) poor concept specialisation capabilities, often including inability to represent exceptions and perform default reasoning, and b) reduction in the naturalness of the object-based representation, since logically defined objects do not really impose any structure to the domain knowledge. Also, most of the systems combine either logic and classes or logic and frames, but not logic and a suitable combination of frames and classes that exploits their strong points.

Finally, an issue not paid much attention, hence not addressed by existing combinations, is the flexibility that a knowledge representation model should provide as far as control knowledge is concerned. This flexibility is important, especially in implementing expert systems, since a single control regime is usually not adequate [1].

In this paper, the main aspects of knowledge representation and reasoning in SILO, a **S**ystem **I**ntegrating **L**ogic in **O**bjects, are presented. SILO is a general purpose hybrid knowledge representation system/language that uses a first-order logic within an object-based framework. Its objects are composed of elements taken from both frame-based and class-based formalisms. SILO also allows user to specify its own inference controls.

The outline of the paper is as follows. In Section 2, the principles integration is based on are discussed. In Section 3, the way knowledge is structured in SILO is presented. In Section 4, the integrated domain knowledge representation language used to represent object internal knowledge is described. Section 5 deals with SILO's inheritance mechanism, whereas Section 6 with SILO's reasoning mechanism. Section 7 is concerned with control knowledge representation in SILO. Section 8 provides some examples mainly illustrating reasoning in SILO. In Section 9 related work is discussed, and Section 10, finally, concludes.

2. Integration Principles

Three basic aspects can be distinguished in a system integrating logic in objects: the knowledge structuring model, the knowledge representation formalism and the control representation model. The *knowledge structuring model* refers to the local structure of objects as well as their global organisation. The *knowledge representation formalism* deals with representation of the object internal knowledge. Finally, the *control representation model* concerns the way control knowledge is represented and applied. The integration principles discussed in this section concern all these aspects.

2.1. The object-based model

There are a number of knowledge representation or programming languages that can be characterised as object-based, such as frame-based, class-based and actor-based languages [3]. From these formalisms, frame-based and class-based languages are more closely related, and these are referred to as object-based representations in this paper. We distinguish three aspects of an object-based representation, namely the classification model, the object representation and the communication mechanism. The *classification model* concerns the global organisation of the objects and their relations, e.g. the specialisation/inheritance scheme. The *object representation* refers to the representation of knowledge within the basic representation unit (an object) of the language. Finally, the *communication mechanism* specifies the way objects can communicate with one another.

Frames and classes have been developed in different environments and for different purposes, namely knowledge representation and object-oriented programming respectively. Thus, although they have a number of obvious similarities, they also have a number of important differences. Frames [2 Ch.6, 3 Ch.8, 4] are more declaratively oriented than classes; a frame has a structure representing a concept via a number of *slots*, each of which is further described by a number of *declarative facets*; a slot may hold a value or be attached local procedures, by means of *procedural facets*, like the if-needed, if-added and if-removed facets, activated on slot access. This is called *procedural attachment*. Classes [3 Ch.2, 5], on the other hand, are more procedurally oriented; a class represents a concept by means of a number of procedures, called *methods*, that specify its behaviour. Methods operate on data stored in variables, called *state variables*, that constitute an object's state. Both frames and classes are organised in a hierarchy, where objects lower down can *inherit* knowledge from objects higher up. A frame-based hierarchy model is more flexible compared to a class-based one, since the former is based on the *prototype theory* and the latter on the *set theory* [3 Ch.7]. Thus, there is a strict distinction between a class object (data structure) and an instance object in a class-based model. An instance cannot differentiate its behaviour (methods) from that of its class; instances of the same class have the same behaviour, but different states. On the contrary, any subframe can differentiate itself from its superframe by having extra slots; also, incomplete frames, that is frames not having values for all of their slots, are allowed at any level in the hierarchy. Classes possess a strong communication mechanism between objects, namely *message passing* (or sending), whereas frames are more autonomous units. Sending a message results in activation of a method, with the arguments conveyed in the message.

We can describe the above two types of object-based representations by the following equations:

class-based model = set theory based hierarchy +
procedural object representation +
message passing +
inheritance

frame-based model = prototype theory based hierarchy +
declarative object representation +
procedural attachments +
inheritance

The main issue here is which elements useful for knowledge representation and reasoning should be taken from frames and which from classes to constitute the object-based model in the integration. To this end, two fundamental elements, one from frames and the other from classes, are employed in the proposed model. The first is the declarative and more expressive structure of a frame. The second fundamental element is message passing, used in classes, by which pieces of knowledge sited in different objects can be interrelated in a way that can play a significant role in reasoning, since computation of the value of a slot may require use of knowledge belonging to other objects.

From the procedural facets, used in frames, only the if-needed facets are embedded in the model. The other types of procedural facets (i.e. if-added, if-removed facets) are not employed in our model. As a consequence, objects have not a state that can be changed during the course of computation (immutable objects). Actually, the notion of if-needed procedures is extended to that of methods, used in classes.

Finally, the specialisation hierarchy relies on a restricted prototype theory and supports multiple inheritance. There is a distinction between classes and instances, but it is not as sharp as in a class-based model. A subclass of a class can have new slots defined in it. An instance can only have new values of slots that substitute for those in its class(es), but not new slots defined in it and cannot be further specialised. This kind of relation is more flexible than the instantiation relation, used in class-based models, but a bit less flexible than that used in frame-based models, where it allows for new slots to be defined in an instance frame as well. However, this restricted specialisation better fits our intuitions about an instance, thus preventing from unintuitive representations.

The combined object-based model can be represented by the following equation:

object-based model = restricted prototype theory based hierarchy +
declarative object representation +
message passing +
inheritance

2.2. The logic-based model

So far, first-order predicate calculus (FOPC) is the most widely used logic-based knowledge representation formalism [6]. Higher order logics are difficult to handle, whereas lower order logics, such as those based on propositional calculus, are expressively very poor. There are also subsets of FOPC, such as Horn-type logics, that are simpler, offer greater efficiency than FOPC, but reduced expressiveness. Apart from its expressiveness, another advantage of FOPC, and of any other logic, is that it comes with a clear and theoretically sound semantics, namely *declarative semantics*.

Any logic is also accompanied by a *proof procedure* for deriving new propositions from a set of already existing propositions. The proof procedure constitutes the operational semantics of a logic-based language and is typically sound and complete. A proof procedure uses one or more inference rules. The most powerful inference rule is *resolution principle* with its associated proof procedure, namely *resolution refutation*. Resolution requires that clausal form of FOPC is used. The above logic-based representation model can be described by the following equation:

logic-based model = FOPC clauses + resolution refutation

Recently, first-order *many-sorted logics* (MSLs) have been paid much attention [7, 8]. A MSL is composed of two integrated components, one for the description of the *signature(s)* and the other logic-based. The description of a signature consists of declarations of the sortal relations between concepts (objects) organised in a lattice-type graph and declarations of the types of the arguments of the predicates used in the logic-based component. Under simple restrictions, a hierarchy of objects (concepts) can be regarded as a lattice-type graph and the arguments type declarations as concepts structural descriptions, so that an MSL-like language can be used for the description of the object internal knowledge. Use of an MSL-based language brings two main advantages. First, it results in simpler (shorter) expressions than unsorted logic, which results in conciseness of representation within an object. Second, the size of the search space for proofs is significantly decreased, thus improving efficiency [7]. The logic-based model used in SILO can be finally described by the following equation:

logic-based model = first-order sorted clauses + sorted resolution refutation

2.3. The integrated model

There are two ways to combine logic and objects into a unified formalism. The first suggests incorporation of object-based notions within a logical framework. This means that the logic-based formalism is (syntactically) extended by new constructs to be able to incorporate notions such as object definition, object classification, message passing etc. This is the way followed by most of the systems that are extensions of the logic programming paradigm, particularly those of Prolog programming, where at the implementational level everything is translated into logical expressions. Thus, a knowledge base remains a flat set of logical expressions. Also, deduction related processes, such as resolution refutation and unification, actually remain unchanged, although operational semantics may be superficially extended.

The second way, followed in SILO, consists in using logic within objects. In this way, logic is used to express object internal knowledge, but not the hierarchical (inheritance/specialisation) relations between objects. Each object is a structured unit, implemented as a separate data structure, that contains the knowledge about itself, mainly expressed as logical expressions. Also, the proof procedure is a tight integration of message passing, inheritance and logical deduction. In this fashion, message passing to an object is regarded as a theorem proving request from the theory (knowledge) related to the object.

A weak point of logic is its great difficulty or inability to efficiently represent purely procedural knowledge. Therefore, our integrated model supports a kind of procedural attachment.

The integrated model used in SILO can be described by the following equation:

integrated model = restricted prototype theory based hierarchy +
sorted logic-based object representation +
procedural attachment +
sorted resolution refutation +
message passing +
inheritance

Our main aim here is the design of an implementable system that gives pre-eminence to objects, thus we are not concerned with issues like soundness and completeness, which should be of main concern in systems the are based on or give pre-eminence to logic. Although soundness and completeness may be desired attributes of an inference system, they often result in restricting its representational flexibility.

CommonLisp [9] is a language suitable for implementing systems like SILO, since it facilitates creation of named data structures with named components for representing objects. Since CommonLisp is the intended implementation language, a Lisp-oriented notation and terminology is used throughout the paper.

2.4. The control model

The advantages of explicit and separate representation of control knowledge have been pointed out by a number of researchers, e.g. [10, 1]. The most well-known and advantageous architecture for implementing such a separation is that offered by *meta-level systems* [11]. This type of architecture provides a separate object-level and meta-level interpreter. Object-level interpreter reasons about domain knowledge, whereas meta-level interpreter reasons about how to use 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.

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 [11].

Programmable steps are implemented in SILO as user-definable functions, called *meta-functions* (for an extensive treatment see [12]). Thus, by (re)defining a number of meta-functions the user can specify a variety of control strategies or heuristics.

3. Knowledge Structuring Model

3.1. Object structure

In common with most object-based languages, objects in SILO are organised in a hierarchy based on an acyclic directed graph model which allows for multiple parents, with the object `object` as its root (see e.g. Fig. 2). Each object corresponds to a concept in the domain described via a number of *attributes* which constitute its conceptual structure. Attributes correspond to state variables of classes or to slots of frames.

The (internal) structure of an object in SILO is depicted in Fig. 1. It consists of three parts: structure-part, knowledge-part and control-part. The *structure-part* of an object accommodates structural knowledge related to the object. It consists of two components. The first, *links*, accommodates taxonomic knowledge related to the object, that is knowledge about its hierarchical relations with other objects. The second, *attributes*, corresponds to declarative facets of frames by including descriptions of the attributes of the object in terms of restrictions on their values. The *knowledge-part* of an object includes knowledge related to the values of its attributes. It consists of two components. The first, *axioms*, includes knowledge about the object expressed in a *declarative* way. This knowledge is distinguished in facts, expressed by *slot-axioms* that correspond to variable-value pairs of classes or slot-value pairs of frames, and in non-factual knowledge, expressed by *method-axioms* (see Section 4.3). The second component, *procedures*, includes non-factual knowledge expressed in a *procedural* way. Method-axioms and procedures correspond to methods of classes or to if-needed procedures of frames. 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 [13, 11]. It consists of two components, namely *deduction-control* and *inheritance-control*. The first component is responsible for controlling the deduction process within the context of the object, e.g. the search strategy to be followed. The second is responsible for controlling the inheritance mechanism, e.g. the inheritance path to be followed.

structure-part
  links
  attributes
knowledge-part
  axioms
  procedures
control-part
  deduction-control
  inheritance-control

Fig. 1. The structure of an object in SILO.

3.2. Objects hierarchy

Two types of an object are distinguished. An *instance-object* (or *instance*) contains knowledge about an individual concept. A *class-object* (or *class*) contains knowledge related to a generic concept. Since the specialisation hierarchy relies on the prototype theory, this knowledge is default knowledge about the individual concepts it represents. Each class, except `object`, is a *subclass* of one or more classes higher up, called its *superclass(es)*. There is a link between a class and each of its subclasses that represents a specialisation/inheritance relation, called a *subclass-of* relation (see Fig. 2), whereas the reverse relation is called a *superclass-of* relation. This means that a (sub)class can differentiate itself from its superclass(es), by a number of ways. A subclass can have new attributes defined in it.

A class can also have instances attached to it, called its *local* (or *own*) *instances*. Instances are terminal nodes in the hierarchy. Classes that have only instances attached to them are called *terminal classes*. There is no terminal class with no instance attached to it. An instance may belong to more than one class. There is a link between a class and each of its instances that represents a *restricted* specialisation/inheritance relation between them, called an *instance-of* relation (see Fig. 2), whereas its reverse relation is called a *class-of* relation. It is restricted in the sense that an instance cannot have new attributes defined in it. This means that an instance has the same conceptual structure as its class(es). Furthermore, instances cannot be further specialised.

We use C_i and O_i as *class symbols* and *instance symbols* respectively, to represent classes and instances in a hierarchy. We also use "<<" and "<" to represent the "subclass-of" and the "instance-of" relations respectively. So, $C_2 << C_1$ means that C_2 is a subclass of C_1 or equivalently that C_1 is a superclass of C_2. The "subclass-of" relation is transitive, that is if $C_3 << C_2$ and $C_2 << C_1$ then $C_3 <<$

C_1. For example, in the hierarchy of Fig. 2, `human` << `mammal` and `mammal` << `animal`, so `human` << `animal`. A child (subclass) of a class is called an *immediate subclass* of it, whereas a parent (superclass) of a class is called an *immediate superclass*. For example, `man` is an immediate subclass of `human`, in contrast to `dad-mimic`, whereas `human` is an immediate superclass of `man`. Also, $O_1 < C_1$ means that O_1 is an instance of C_1 or equivalently that C_1 is a class of O_1. For example, `m2` < `man`, `m3` < `writer` and `w2` < `woman`. Finally, by I_{C_i} and D_{C_i} we represent the sets of local instances and the immediate subclasses of C_i respectively. For example, I_{man} = {`m2, m3`} and D_{human} ={`man, writer, woman`}.

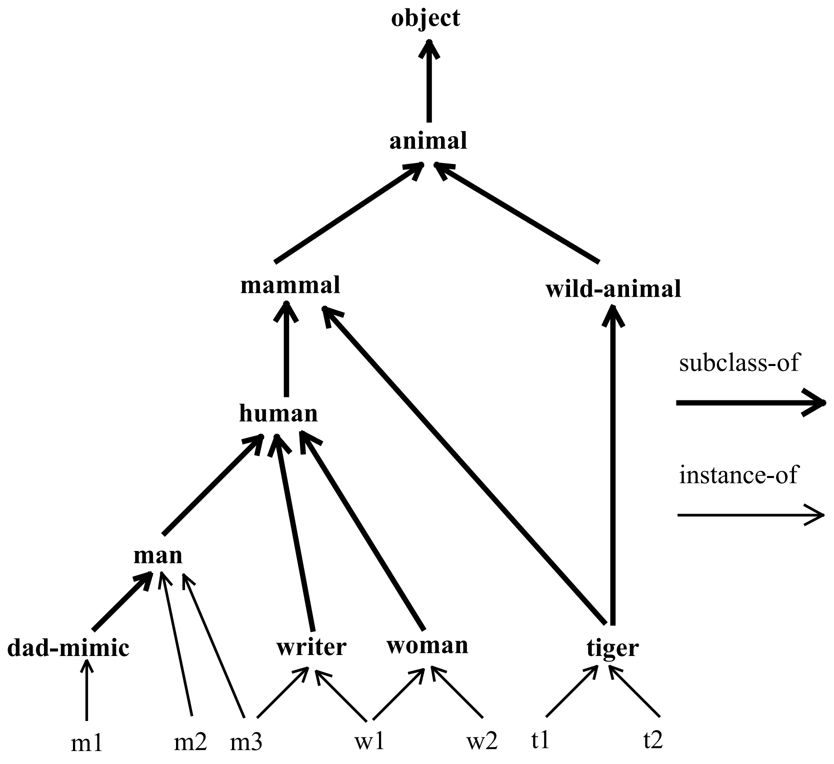

Fig. 2. A (partial) hierarchy of objects.

Two classes are (declared to be) *disjoint* if they (should) have no common instances. For example, in Fig. 1, `man` and `woman` are (declared to be) disjoint.

4. Domain Knowledge Representation

The SILO knowledge representation language concerns both domain and control knowledge. Each SILO expression is an *object definition* and results in the creation of an object with a structure like that of Fig. 1. There are two types of an object definition, namely class definition and instance definition.

A *class definition* has the format,

```
(defclass   <class symbol>
             <class-structure-decls>
             <knowledge-decls>
             <control-defs>)
```

whereas an *instance definition* the format,

```
(definst   <instance symbol>
            <inst-structure-decls>
            <knowledge-decls>
            <control-defs>).
```

In the definitions, <class-structure-decls> and <inst-structure-decls> concern the structure-part of an object, <knowledge-decls> its knowledge-part and <control-defs> its control-part. Each of these templates has its own language.

The language for representation of the domain knowledge of an object is an integration of two component languages. The one, called *structure declaring language* (SDL), concerns the structure-part template in an object definition and the other, called *message passing logic* (MPL), concerns the knowledge-part template. The integrated language can be considered as an extended form of a *many-sorted logic*. SDL expressions correspond to the description of a signature in a MSL, whereas MPL corresponds to the logic-based component of a MSL. In the following subsections, SDL and MPL are presented. The language for the representation of the control knowledge (control-part) is discussed in Section 7.

4.1. Structure declaring language (SDL)

SDL is used for the description of the structure-part of an object, via the structure-part template in an object definition:

```
<class-structure-decls> ::=  <class-link-decls>
                             <attribute-decls>
                             <definition-decls>

<inst-structure-decls> ::=  <inst-link-decls>
                            <attribute-decls>
```

So, SDL provides three types of declarations, namely link declarations, attribute declarations and definitional declarations, discussed in the sequel.

4.1.1. *Link declarations*

The first type of declarations, *link declarations*, describe the conceptual links of the object and are stored in 'links'. Link declarations are defined as follows[a] :

[a] The symbol * means that there can be zero, one or more occurrences of the expression at its left.

<class-link-decls> ::= ((<inher-link-decl>*)
(<disj-link-decl>*))

<inst-link-decls> ::= ((<inher-link-decl>*))

<inher-link-decl> ::= class symbol

<disj-link-decl> ::= class symbol

So, link declarations in a class consists of a number of inheritance link declarations, that are class symbols representing the immediate superclasses of the class, and a number of disjointness link declarations, that are also class symbols representing the classes which are disjoint with the class. Link declarations in an instance consists only of inheritance link declarations, representing its classes.

4.1.2. *Attribute declarations*

The second type of declarations, *attribute declarations*, specify the attributes of an object and represent restrictions on their values. They are stored in 'attributes'. 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 the attribute is a *simple attribute*, otherwise it is a *composite attribute*. An attribute is *single-valued* if it is allowed to take only one n-tuple as its value, otherwise it is *multi-valued*. The name of each attribute is unique.

Attribute declarations are defined as follows:

<attr-decls> ::= (<attr-decl>*)

<attr-decl> ::= (<type-part> <num-part>)

<type-part>::= (b^n t_1 ... t_n)

<num-part>::= (n_{min} n_{max}).

As it is clear, an attribute declaration consists of two parts. In the first, called the *type restriction part*, b^n is an *attribute symbol*, representing an n-place attribute, and t_i (i = 1...n) are the *types* of the component values of the attribute. Each t_i can be either a Lisp data-type predicate or a class symbol or an *enumeration set*. While a Lisp predicate and a class symbol represent the type of the corresponding component value by intention, an enumeration set gives it by enumerating all of its possible values, that is by extenuation. The elements of an enumeration set can be instance or individual symbols. The most general type is denoted by 'object'. In the second part, called the *number restriction part*, n_{min} and n_{max} are positive integers representing the minimum and maximum number respectively of the values (n-tuples) the attribute is allowed to take. If one of these numbers is to be left unspecified, which means "unrestricted", an '!' is put in its place. If both numbers are to be left unspecified, the number restriction part is omitted. Obviously, n_{min} = n_{max} = 1 declares a single-valued attribute.

For example, the attribute declarations **`((member human) (2 5))`** and **`((leader man) (! 1))`** in the class `small-team` denote that, "a small team has at least two and at most five members that are humans" and "a small team has at most one leader who must be a man" respectively. Also, **`((son man))`** and **`((sex (male female)) (1 1))`** in `human` denote that, "a human has a number of sons that are men" and " a human has a sex which is one of 'male' and 'female' " respectively.

The type restriction parts of attribute declarations correspond to declarative facets of a frame-based language or to predicate declarations of a MSL. The number restriction part is similar to the cardinality facet of a slot used in frame-based systems [4]. This kind of restriction can also be found in the terminological component of logic-based hybrid systems, like e.g. BACK [14], as a number restriction on roles of concepts.

4.1.3. *Definitional declarations*

The last type of declaration, *definitional declarations*, represent definitional specifications for a class-object. Definitional declarations are defined as follows:

<defin-decls> ::= (<defin-decl>*)

<defin-decl> ::= ($D\ b_1^{n1}\ \ldots\ b_m^{nm}$)

where D is a built-in *definitional primitive* and each $b_i^{n_i}$ is an attribute symbol. There are two definitional primitives, 'vess' and 'class'. The first denotes that the associated attributes are value *essential attributes*, that is their values are not allowed to be changed lower down, in contrast to *incidental attributes*. The second denotes that they are *class attributes*, that is attributes referring to the class-object as a whole, not applicable to its subclasses or instances, in contrast to *instance attributes*. Use of class attributes is not discussed in this paper.

Introduction of the distinction between essential and non-essential attributes is motivated by the issues addressed in [15], concerning the definitional inability of object-based formalisms. One of the Brachman's points in [15] is that, due to the fact that most of the object-based formalisms do not distinguish between cancellable (incidental) and uncancellable (essential) properties, one can in principle create an instance of a class that overrides all the properties stored in the class. This is not allowed in SILO. For example, if in a class `german-car` the attribute 'country-of-make' holding the value 'germany' is declared as value essential, then it is assured that there would not be any instance of that class that has a different value for that attribute.

Also, introduction of the distinction between class and instance attributes corresponds to the distinction between the own and member slots in frame-based languages [4 Ch.8, 2 Ch.6]. Class attributes of a class are not inherited by its subclasses and instances.

4.2. Message passing logic (MPL)

MPL is used for description of the knowledge-part of an object. MPL is a variant of first-order predicate calculus (FOPC). The template describing the knowledge-part of an object has the following format:

```
<knowledge-decls> ::= <axioms>
                      <procs>

<axioms> ::= (<axiom>*)

<procs> ::= (<proc-def>*)
```

where each <axiom> is an MPL formula and each <proc-def> is a *procedure definition* associated to the formulas in <axioms> for procedural attachment purposes (see Section 4.2.2). In the following MPL is presented.

4.2.1. *MPL formulas*

The primitive structural unit of an MPL formula is an MPL *atom*, which has the same format as in FOPC.

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

The last argument (t_o) always denotes an object and is therefore called the *object argument*. The predicate symbol p^{n+1} represents an attribute of the object denoted by the object argument. Thus, for each n-place attribute there is a homonymous *(n+1)*-place predicate and vice versa. In other words, the set of attribute symbols is identical to that of predicate symbols, except for arity numbers:

$$\forall i,\ b_i^n \equiv p_i^{n+1}.$$

The arguments t_1,..., t_n represent component values of the predicate's homonymous attribute, and are therefore called *value arguments*. Consequently, attribute declarations also represent restrictions on the arguments of the predicates. In the same way as with attributes, we distinguish between single-valued and multi-valued predicates.

There are three types of non-variable terms in MPL, namely constants, evaluable terms and procedure terms. A *constant* is either an instance symbol or an individual symbol which is member of an enumeration set.

Evaluable terms are divided in two types, namely normal evaluable terms and typed evaluable terms. A *normal evaluable term* has the form

$$!b_i^1$$

(e.g. `!son`). An evaluable term represents the value(s) of the corresponding simple attribute. A *typed evaluable term* has the form

$$!b_i^1 : C_i$$

(e.g. **!son:writer**). It represents the value(s) of the attribute which are of type C_i. Introduction of evaluable terms aims only at the conciseness of representation. So, they are expanded before they are used in a reasoning process (see Section 3.2.4).

Variables are actually *typed* (or *sorted*)[b] 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'. In the latter case it is called a *universal variable*. Finally, there is the *special variable* '?self ' which has a special semantics, specified in the subsequent sections.

A *procedure term* has the form

$$\#(f^n\ t_1 \ldots t_n)$$

where f^n is a (user-defined or built-in) *computable function symbol*, and $t_1, \ldots, t_n$ its arguments that are terms (e.g. **#(compute-weight !height)**, **#(compute-allowance ?x)**).

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

Definition of an MPL non-atomic formula is identical to that of FOPC. Cambridge Polish notation is used (see below) with as connectives {**~** , **&** , **V** , **=>**} for {negation, conjunction, disjunction, implication} and quantifiers {**forall**, **exists**}. Full syntax of FOPC without function expressions is available (see [2 Ch.3, 6 Ch.2] for the basic FOPC terminology). The special variable **?self** is regarded as a universally quantified variable in the context of the class it is used, and, unless otherwise specified, is assumed to have wide scope. We therefore, in general, omit its quantifier.

Definition 3. (MPL formula) A well-formed MPL formula (wff) is defined as follows:
(i) an atom is a wff.
(ii) if F is a wff, (**~** F) is also a wff.
(iii) 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.
(iv) if F is a wff and v_i a free variable in F, ((**forall** v_i) F) and ((**exists** v_i) F) are also wffs.
(v) nothing else is a wff.

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

MPL does not directly support standard FOPC function expressions. However, SILO can indirectly use arbitrary functions by declaring them as single-valued

[b] We use 'typed' as synonymous to 'sorted', used in many-sorted logics.

attributes and asserting their values via MPL formulas. For example, to express **(father-of john) = peter**, given that `john` and `peter` are instances of `man`, **((father man) (1 1))** is declared in `human` and the formula **(father peter john)** is introduced in `john`.

4.2.2. *Procedural attachment*

MPL supports a kind of procedural attachment. Procedural attachment is achieved via procedure terms. Definitions of computable functions related to an object are represented as procedure definitions in the object's definition and are stored in its 'procedures' component.

A *procedure definition* has the following structure, which is similar to a Lisp function definition:

```
<proc-def> ::= (<fun-name> (<args>) <body>)

<fun-name> ::= computable function symbol

<args> ::= (<arg>*)

<arg> ::= MPL term

<body> ::= (<p-expr>*)

<p-expr> ::= Lisp expression
```

Apart from Lisp built-in primitive functions a SILO built-in function, namely *call-super*, can be used in <body>. It takes as arguments a computable function name with its corresponding arguments; it applies a procedure, from a superclass of the object, with the name specified to the arguments specified. The superclass is the first met in the class precedence list (see Section 5.3) after the object. This corresponds to the 'super' facility of class-based systems [3, 5].

The way procedural attachment is implemented here is different from the standard way (e.g. [16], Prolog), where evaluable/computable predicates are used rather than computable terms. Our way of incorporating procedural attachment gives a more natural representation, corresponding to the attribute-value model of object-based systems. Myers' universal attachment mechanism [17] also allows for procedural attachments to terms of a logical formula.

4.2.3. *Formula expansion*

MPL is an extension of FOPC as far as introduction of new types of terms is concerned, such as normal and typed evaluable terms, typed variables and procedure terms, which are called *extensions*. From them, evaluable terms are only used for syntax conciseness purposes. Therefore, MPL formulas containing evaluable terms are expanded to equivalent MSL-like formulas at creation time. To this end, each evaluable term has a corresponding *expansion*, presented in Table 2, where v_i is a variable and t_o represents the object where the formula containing the term is stored in. t_o is either the symbol of the object, if it is an instance, or the special variable '?self ', otherwise.

An MPL formula is expanded according to the rules presented in Table 1. In that table, Q_U and Q_E represent universal and existential quantification respectively and F represents an MPL expression. F_N is a conjunction with as conjuncts the expansions of the evaluable terms occurring in F. F_T is F with its extensions been replaced by the (new) variables introduced. Finally, Q_{UT} is a universal quantification binding the new variables.

Table 1. Expansion Rules

Initial MPL formula		Expanded MPL formula
$(Q_U\, F)$	$\longrightarrow$	$(Q_{UT}\, (\Rightarrow F_N\, F_T))$
$(Q_E\, F)$	$\longrightarrow$	$(Q_{UT}\, (\Rightarrow F_N\, (Q_E\, F_T)))$

For example, **`((forall ?x) (=> (son !son ?x:woman) (wife ?x ?self)))`** in `man` is expanded to

```
((forall ?x ?v)
  (=> (son ?v ?self)
      (=> (son ?v ?x:woman) (wife ?x ?self)))).
```

Table 2. Expansions and Translations

extension		expansion		translation
$!b_i^1$	$\longrightarrow$	$(p_i^2\ v_i\ t_o)$	$\longrightarrow$	$(p_i^2\ v_i\ t_o)$
$!b_i^1{:}C_i$	$\longrightarrow$	$(p_i^2\ v_i{:}C_i\ t_o)$	$\longrightarrow$	$(\&\ (C_i\ v_i)$ $(p_i^2\ v_i\ t_o))$
$v_i{:}C_i$	$\longrightarrow$	-----	$\longrightarrow$	$(C_i\ v_i)$

Also, **`((exists ?x) (gives !son ?x:gift ?self))`** in `man` is expanded to

```
((forall ?v)
  (=> (son ?v ?self)
      ((exists ?x) (gives ?v ?x:gift ?self)))).
```

This expanded form of MPL formulas is used during a reasoning process.

4.2.4. *Semantics*

In this section, a declarative semantics for MPL is provided, by giving a translation of MPL formulas into FOPC formulas. Because definition of a non-atomic formula in MPL is identical to that of FOPC, translation only concerns the introduced extensions, except procedure terms that cannot be given a declarative semantics. To this end, each extension has a *translation*, presented in Table 2.

An MPL formula is translated into a FOPC formula according to the rules presented in Table 3. The symbols Q_U, Q_E, Q_{UT} and F_T have the same semantics as in table 1. F_N is a conjunction with as conjuncts the translations of the extensions occurring in F. F_{N1} and F_{N2} are conjunctions with as conjuncts the translations of the evaluable terms and the typed variables in F being under the scope of Q_E respectively.

Table 3. Translation Rules

MPL formula		FOPC formula
$(Q_U\ F)$	——————>	$(Q_{UT}\ (\Rightarrow F_N\ F_T))$
$(Q_E\ F)$	——————>	$(Q_{UT}\ (\Rightarrow F_{N1}\ (Q_E\ (\&\ F_{N2}\ F_T))))$

If the translated formula contains the special variable '?self ', then a further step is required, based on the first of the above rules: '?self ' is treated as a typed variable with as type the class the formula is stored in.

The example formulas used in the previous section are translated into

```
((forall ?v)
   (=> (man ?v)
       ((forall ?v1 ?v2)
          (=> (& (son ?v1 ?v) (woman ?v2))
                 (=> (son ?v1 ?v2) (wife ?v2 ?v))))))
```

and

```
((forall ?v)
    (=> (man ?v)
        ((forall ?v1)
            (=> (son ?v1 ?self)
                ((exists ?v2) (& (gift ?v2)
                                 (gives ?v1 ?v2 ?self)))))))
```

respectively. From the examples, it is quite clear that an MPL expression is much shorter than the corresponding FOPC one. This is basically due to the use of typed and evaluable terms.

4.2.5. *MPL Clausal form*

The clausal form of an expanded MPL expression is produced using the same standard procedure as in FOPC, except for some simple modifications. First, the type of a variable is retained. Second, procedure terms are treated as constants. Finally, two new terms are introduced, namely *typed Skolem constants*, and *typed Skolem functions*. For example, the clausal form of the MPL expression

```
((exists ?x) (P ?x:C ?self))
```

is

```
((P (skf1 ?self):C ?self)),
```

whereas that of

```
((exists ?x) ((forall ?self) (P ?x:C ?self)))
```

is

```
((P skc1:C ?self)),
```

where **skf1**, **skc1** are a skolem function and a skolem constant symbol respectively.

A literal is *ground* if its value arguments are ground terms. Ground terms are: constants, Skolem constants, ground Skolem function expressions, the special variable '?self '. A Skolem function expression is ground if its arguments are ground terms.

Because of the attribute declarations, the notion of *well-typedness* is introduced, as in MSLs. An MPL atom is *well-typed* if each of its value arguments is well-typed, otherwise it is *ill-typed*. A value argument in an MPL atom is well-typed if it is compatible with the corresponding type in the declaration of the homonymous attribute of the atom's predicate. Well-typedness checking takes place at creation time. Well-typedness plays a role in reasoning: ill-typed literals are not considered for resolution.

Three types of literals are distinguished, namely ordinary literals, procedure literals and message literals. A *procedure literal* is a literal that contains at least one procedure term.

A *message literal* is a literal that contains no procedure terms and its object argument is

(i) a constant representing an object different from that the axiom of the literal is stored in or

(ii) a typed variable,

that is it denotes an object different from the current object which is called the *receiver*. A message literal is denoted by $L_i^{R_i}$, where R_i represents the receiver. A clause containing only message literals is called a *message clause*. Any other literal is an *ordinary literal*. Ground literals are ordinary literals.

4.2.6. *Message passing*

The only way that objects can communicate between each other is by message passing, which is a fundamental mechanism/operation in SILO. A *simple message* is a message clause containing message literals of the same receiver:

$$M_i^{R_i} = (L_1^{R_i} \ \ldots \ L_n^{R_i}).$$

The semantics of a simple message passing is to prove the body of the message, the message clause, in the context of the receiver (see next section).

A *compound message* is a message clause consisting of message literals that have not the same receiver. A compound message is organised as a sequence of simple messages:

$$\boldsymbol{M} = (M_1^{R_i} \ \ldots \ M_n^{R_i}).$$

The semantics of a compound message is to conjunctively prove the bodies of its simple messages in the contexts of the corresponding receivers: if any simple message fails, the compound message also fails (see next section).

If the object argument in a message literal is a typed variable, it is a *class message*, otherwise it is an *instance message*. The receiver of a class message is a class-object, whereas of an instance message one or more specific instance-objects. An instance message corresponds to the traditional 'extensional' message used in class-based languages. A class message is a generalisation of the 'intentional' or 'anonymous' [18] or 'broadcast' [19] message. When the receiver is the object `object`, then a class message is identical to an intentional message. For example, **`(plays ?x george)`** in **`(=> (plays ?x george) (plays ?x john))`** stored in `john` is an instance message, whereas **`(son ?y ?x:woman)`** in **`(=> (& (son ?y ?self) (son ?y ?x:woman)) (wife ?x ?self))`** stored in `man` is a class message. (We use non-clausal form in the examples and omit quantifiers for readability reasons).

4.3. Types of axioms

Because SILO uses a resolution-based inference mechanism, expanded MPL formulas are converted into clausal form at creation time. Thus, the knowledge-part of an object includes a number of (expanded) MPL clauses, called *axioms*, stored in the 'axioms' component. There are two types of axioms (clauses), namely *slot-axioms* and *method-axioms*. We say that an axiom *belongs to* an object if it is stored in that object.

4.3.1. *Slot-axioms*

A slot-axiom is an MPL unit clause (axiom). Its object argument is either the symbol of the instance-object it belongs to, e.g. **`(son mike john)`** and **`(~ (plays ?x:wind-instrument john))`** in `john`, or the special variable '?self', if it belongs to a class-object, e.g. **`(sex male ?self)`** in `man` and **`(eats ?x:meat ?self)`** in `human`. A slot-axiom represents a fact or a number of facts about an attribute of an object and can be viewed as an attribute-value expression. A multi-valued attribute gives rise to more than one slot-axiom with the same predicate. SILO in general allows for n-ary predicates.

The restriction that the last argument in an MPL atom always denotes an object gives a slightly different interpretation of it. Thus, the slot-axiom **`(likes mary john)`** represents the fact "John likes Mary" and not the fact "Mary likes John" as usual. Reversely, the fact "John eats beef" is represented as **`(eats beef john)`** and not as usual **`(eats john beef)`**, since this piece of knowledge refers to John, that is 'eats' is an attribute of `john` and 'beef ' its value.

There are two non-overlapping groups of slot-axioms, namely *class axioms* and *instance axioms*. The former include axioms whose predicate's homonymous attribute is a class attribute, whereas the latter those corresponding to instance attributes. Instance axioms are further divided in two other groups, namely *essential axioms* and *incidental axioms*. The former are slot-axioms whose predicate's homonymous attribute is an essential attribute, whereas the latter correspond to incidental attributes. Use of class axioms is not discussed in this paper.

4.3.2. *Method-axioms*

Method-axioms are non-unit MPL clauses (axioms) and represent non-factual knowledge relating to the attributes of an object. Method-axioms correspond to methods in class-based languages or to if-needed procedures in frame-based languages. Method-axioms may carry messages.

We distinguish three types of axioms: ordinary-axioms, message-axioms, and procedure-axioms. An *ordinary-axiom* is an axiom containing only ordinary literals. An ordinary-axiom only deals with local knowledge. For example, **`(=> (& (inp1 0 ?self) (inp2 0 ?self)) (out 0 ?self)))`** in `or-gate` is an ordinary-axiom. A *message-axiom* is an axiom containing at least one message literal, but no procedure literals. Message axioms carry messages. A message axiom deals with knowledge related to other objects as well. For example, **`(=> (& (son ?y ?self) (son ?y ?x:woman)) (wife ?x ?self))`** in `man` is a message-axiom.

A *procedure-axiom* is an axiom containing exactly one procedure literal, and any number of message or ordinary literals. It is actually a Horn-type axiom whose head is a procedure literal. A procedure-axiom may involve knowledge related to other objects, since it may include message literals. For example, **`(=> (height ?y ?self) (weight #(compute-weight ?y) ?self))`** and **`(=> (& (father ?x ?self) (income ?y ?x)) (allowance #(compute-allowance ?y) ?self))`** are procedure-axioms. Each procedure-axiom is relating to a single-valued attribute. Thus, a procedure-axiom represents one procedure (in case of a simple attribute) or a set of procedures (in case of a composite attribute) that computes the value of a single-valued attribute. The variables in a procedure term of the procedure literal of a procedure-axiom are assumed to be instantiated via the body of the procedure-axiom. A similar idea is used in RHET [20], in that variable arguments in a function call may have been instantiated via a previously performed unification, but not during the current unification, as in SILO.

5. Inheritance in SILO

Inheritance is a fundamental mechanism in SILO as in most object-based systems. 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. In such cases, an instance/class inherits all the knowledge from within its classes/superclasses In cases where conflicting knowledge exists problems arise [21]. In such cases, in order to resolve conflicts, the more specific information about an attribute invalidates the less specific one. The problem is then two-fold, how to detect conflicting knowledge and how to determine its most specific occurrence.

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.

Apart from various inheritance aspects, SILO supports a variety of *specialisation types* between axioms, such as *addition*, various types of *extension*, *substitution*, *refinement* and *exception* of knowledge. In the following subsections we present basic issues of the inheritance mechanism of SILO in a simplified way. For a detailed and formal treatment refer to [22].

5.1. Complete inheritance

A conflict, from the point of view of complete inheritance, is defined as follows, where a_h represents an axiom stored higher up and a_l an axiom stored lower down in the context of an object.

> **Definition 4.** (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 an exception to a_h.

Detection of conflicts then relies on the following definitions concerning the specialisation types involved in Def. 4. In the following, $G(a_i)$ represents the ground instantiations of the axiom a_i.

> **Definition 5.** (Substitution) An axiom a_l is a *substitution* for an axiom a_h if they are definitions for the same single-valued attribute.

> **Definition 6.** (Refinement) An axiom a_l is a *refinement* of an axiom a_h if $G(a_h) \supset G(a_l)$.

> **Definition 7.** (Exception) An axiom a_l is an *exception* to (or *inconsistent* with) an axiom a_h if $G(a_l) \supseteq G(\sim a_h)$.

Based on the above definitions, a few detection theorems for detecting conflicts have been introduced [22]. The examples below, that refer to the knowledge base of

Fig. 3, give a flavour of the specialisations as well as the detection rules employed in SILO. In Fig. 3, human is a subclass of mammal and john, peter are instances of human, and for the sake of conciseness formulas are in non-clausal form and unexpanded.

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

human
(num-of-legs 2 ?self)
(eats ?x:meat ?self)
(lives-in !works-in ?self)

john                                  peter
(father peter john)                   (eats pork peter)
(~ (likes swimming john))             (likes chess peter)
(~ (eats ?x:anim-prod john))          (=> (lives-in ?x john) (lives-in ?x peter))
(=> (likes ?x !father) (likes ?x john))
```

Fig. 3. An Example Knowledge Base in SILO.

So, (~ (likes swimming john)) in john is an *exception* to (or inconsistent with) (likes swimming ?self) in mammal. Also, (~ (eats ?x:animal-product john)) in john is an exception to (eats ?x:meat ?self) in human, since meat is a subclass of animal-product (Def. 7). As it is clear, negation is used to express exceptions. This representation scheme, called *exception by negation*, is very powerful in representing knowledge, like e.g. representing state changes in planning problems in the blocks world [22].

On the other hand, (num-of-legs 2 ?self) in human is a *substitution* for (num-of-legs 4 ?self) in mammal, since they are definitions for the same attribute 'num-of-legs'. Similarly, (=> (lives-in ?x john) (lives-in ?x peter)) in peter is a substitution for (lives-in !works-in ?self) in human (Def. 5). Finally, (eats pork peter) in peter is a *refinement* of (eats ?x:meat ?self) in human, since pork is an instance of meat (Def. 6).

Apart from the above types, SILO also supports other types of specialisation that do not create conflicts. For example, (eats ?x:meat ?self) in human is an *extension* of (eats ?x:vegetable ?self) in mammal, since it introduces new values for a multi-valued attribute. Also, (lives-in !works-in ?self) is an *addition* to knowledge in mammal, since it introduces knowledge about a new attribute.

As far as inheritance of attribute declarations is concerned, any attribute declaration lower down is conflicting with any attribute declaration higher up that refers to the same attribute.

After detection of conflicts, *overriding* (or *masking*) is employed in SILO, as in object-based systems [3, 5]. This means that an axiom a_l lower down overrides any conflicting axiom a_h stored higher up in the hierarchy. The same holds for attribute declarations.

Apart from the built-in specialisations and detection rules, the user can define its own by (re)defining the inheritance meta-functions (see Section 7.1 and the Appendix), stored in the corresponding objects.

5.2. Atomic inheritance

The above principles and techniques concern complete inheritance. However, they are not appropriate to deal with cases where not all of the atomic consequences of axioms are inherited. Therefore, another technique, called *consequence retraction*, is employed in SILO. Consequence retraction is similar to 'solution invalidation' (notion introduced in [23]). We use a more general view of this technique:

> **Definition 8.** (Consequence Retraction) An atomic consequence found during a reasoning process in the context of an object is retracted if it is inconsistent with any of the axioms in the object's current theory.

For example, the answer to query **`((exists ?x) (likes ?x john))`** would be **`(likes swimming john)`** and **`(likes chess john)`**, since John likes whatever his father likes (see Fig. 3). However, the answer is only **`(likes chess john)`**, because the other candidate answer is invalidated, as being inconsistent with **`(~ (like swimming john))`**. Again here negation is used to represent exceptions to defaults.

5.3. Inheritance order

If the conflicting axioms are not within classes that belong to the same path/branch in a hierarchy, but within classes/superclasses of an instance/class that belong to different paths, the situation is more complicated. In these cases inheritance order should be also considered. 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 a precedence list of the (super)classes of any object. The *class precedence list* ι_{O_i} of an object O_i is an ordered set of the (symbols 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, to determine the inheritance path. The user, however, can define its own strategy for determining the inheritance path either globally or locally via the inheritance meta-functions. The precedence list ι_{O_i} of each instance-object is computed at creation time. For example, the precedence list of `m2` in the hierarchy of Fig. 2, based on the default strategy, is ι_{m2}=`{man, writer, human, mammal, animal, object}`.

6. Reasoning in SILO

In this section, we discuss SILO's operational semantics, that is how reasoning is performed in SILO. The inference mechanism of SILO is a tight integration of inheritance, resolution refutation and message passing. A proof process starts off as soon as a message is sent to an object. Message passing to an object in SILO means sending an MPL clause, called a *theorem*, to be proved from the theory available in the context of the object. The *context* of an object is defined differently for an instance and a class. The context of an instance O_i , is defined as the union of its local theory and the theories it can inherit from objects higher up. The *local theory* $\llcorner_{O_i}$ of an object O_i includes the axioms locally stored in O_i . The context of a class is defined as the union of the contexts of its instances and subclasses.

6.1. Proof process

Each query, that is an MPL formula F set by the user, is negated and converted into its clausal form T, which is considered as a message. If it is a simple message, it is sent to the corresponding receiver (object) to be proved. If it is a compound message, its constituent simple messages are successively sent to the corresponding receivers (objects) to be proved. If any partial proof fail, the whole proof (query) fails. This is called the *query-based proof process*, which is the top-level process, and is more formally described below. Each partial proof process is an *object-based proof process*. In the following, M_c and R_c represent the current simple message to be processed and the corresponding receiver respectively, whereas $\mathcal{P}_{R_c}(M_c)$ denotes an object-based process. Also, **`first`** and **`rest`** are two functions acting in the same way as the corresponding Lisp functions, whereas **`receiver`** is a function that specifies the receiver of a simple message clause.

(1) Negate F and convert it to its clausal form T; set $T_c = T$.
(2) If T_c is a simple message, set $M_c = T_c$ and $R_c =$ **`receiver`**(T_c); goto step 5.
(3) Organise T_c as a sequence of simple messages.
(4) Set $M_c =$ **`first`**(T_c), $R_c =$ **`receiver`**(M_c) and $T_c =$ **`rest`**(T_c).
(5) If M_c is a ground message, check consequence retraction in $\llcorner_{R_c}$.
(6) If M_c is retracted, stop (failure).
(7) Perform $\mathcal{P}_{R_c}(M_c)$.
(8) If $\mathcal{P}_{R_c}(M_c)$ is not successful, stop (failure).
(9) If T_c is the empty clause, stop (success).
(10) Goto step 4.

There are different object-based proof processes followed in the case of an instance and a class message, called *instance-based proof process* and *class-based proof process* respectively. An instance-based proof process takes place in the context of an instance, whereas a class-based one in the context of a class.

6.1.1. *Instance-based proof process*

When a message clause M_i, called a *theorem*, is sent to an instance O_i , the system will first try to prove it using $\llcorner_{O_i}$. If this fails, axioms from its class(es) are inherited, according to their precedence and the inheritance rules, joined to those already available, and another resolution attempt is made. If this also fails, then axioms from the immediate superclass(es) of its class(es) are inherited, joined and tried for resolution, and so on. This process continues until either the query (theorem) is successfully answered (resolved), or there are no more axioms to be inherited, in which case it fails.

The above instance-based procedure is more formally described below, where by O_c and C_n the current (instance) object and the next (class) object in the hierarchy to be considered for inheritance are respectively denoted. The available axioms at any time for proving the *current theorem* T_c constitute the *current theory*, denoted by S_c. The object in whose context reasoning is currently taking place is called the *current object*. The special variable '?self ' is always bound to the symbol of the current object. Finally, by $\mathcal{P}(S)$ an MB-Resolution process in S (see Section 6.3) is denoted. $\mathcal{P}(S)$ is successful, if the empty clause is produced.

(1) Set $O_c = O_i$, $T_c = M_i$ and $S_c = \mathcal{L}_{O_i} \cup \{T_c\}$; start $\mathcal{P}(S_c)$.
(2) If $\mathcal{P}(S_c)$ is successful, check consequence retraction in S_c and stop (success).
(3) Set C_n = **`first(`** ℓ_{O_c}) and ℓ_{O_c} = **`rest(`** ℓ_{O_c}) .
(4) Make $S_c = S_c \cup \mathcal{F}(\mathcal{L}_{C_n})$; start $\mathcal{P}(S_c)$.
(5) If $\mathcal{P}(S_c)$ is successful, check consequence retraction in S_c and stop (success).
(6) If ℓ_{O_c} is empty, stop (failure).
(7) Go to step 3.

$\mathcal{F}(S)$ represents the action of the inheritance rules, both built-in and user-defined. It takes as input the local theory of an object and gives as output the axioms that will be inherited, that is added to the current theory. In cases where there are no conflicting axioms, hence the whole local theory is inherited, a *monotonic extension* of the current theory takes place, otherwise a *nonmonotonic extension* is performed, which facilitates *default reasoning*.

6.1.2. *Class-based proof process*

When a theorem M_i is sent to a class C_i , the system first considers the union of its own essential and the essential slot-axioms of all of its superclasses. If the theorem resolves with any of them and the empty clause is produced, a solution has been found. Otherwise, the theorem is sent to the instances of the class successively. If the empty clause is produced in the context of any instance, then a solution has been found. If not and the class is a terminal class, it fails. Otherwise, the theorem is sent to each of the immediate subclasses of the class successively. In each subclass only its essential slot-axioms are considered, and so on. During this process, if the theorem resolves with an essential axiom in a class, only instances and subclasses of that class are considered afterwards.

The above class-based proof procedure is more formally described below, where the following notation is used. C_c is the current (class) object, in the context of whose reasoning is taking place. A_{C_i} is a set including the essential axioms of C_i and the essential axioms inherited from all the superclasses of C_i. A_c is the current set of essential slot-axioms. $\mathcal{P}(A_i\,,\,T_i)$ denotes a linear resolution process in $(A_i \cup T_i)$, with T_i as the top sentence. A resolution process is *successful* if the empty clause is produced. It is *partially successful* if the empty clause cannot be produced, but there is at least one resolution step performed. A resolution process *fails*, if there is no resolution step performed. Q_f is a flag that can take on one of four values: NIL, SUC, RES and FAIL, representing its initial value, a resolution process success, a resolution partial success and a resolution failure respectively. Also, r_c is the most recently produced resolvent. Finally, G_c is an auxiliary set used for reset purposes. Also, recall that by I_{C_i} and D_{C_i} the instances and the subclasses respectively of C_i are represented.

```
 (1) Set $C_c = C_i$ , $T_c = M_i$ , $Q_f$ = NIL and $G_c$ to be empty.
 (2) Make $A_c = A_{C_i}$ ; start $\mathcal{P}(A_c\,,\,T_c)$.
 (3) If $\mathcal{P}(A_c\,,\,T_c)$ is successful, stop (success).
 (4) Set $I_c = I_{C_c}$ and $D_c = D_{C_c}$.
 (5) If $I_c$ is empty, goto step 9.
 (6) Set $O_c$ = first($I_c$) and $I_c$ = rest($I_c$) ; start $\mathcal{P}_{O_c}(T_c)$.
 (7) If $\mathcal{P}_{O_c}(T_c)$ is successful, set $Q_f$ = SUC.
 (8) Goto step 5.
 (9) If $Q_f$ = SUC, stop (success).
(10) If $C_c$ is a terminal class and $G_c$ is empty, stop (failure).
(11) If $Q_f$ = RES, set $D_c = D_{C_c}$ and $Q_f$ = NIL.
(12) If $Q_f$ = FAIL, set $D_c = D_{C_c}$ ; if $D_c$ is empty, goto step 19.
(13) Set $C_c$ = first($D_c$) and $D_c$ = rest($D_c$) .
(14) Make $A_c = A_{C_c}$ ; start $\mathcal{P}(A_c\,,\,T_c)$.
(15) If $\mathcal{P}(A_c\,,\,T_c)$ is successful, stop (success).
(16) If $\mathcal{P}(A_c\,,\,T_c)$ is partially successful, set $T_c = r_c$ , $I_c = I_{C_c}$ , $Q_f$ = RES
       and $G_c$ to be empty; goto step 5.
(17) If $D_c$ is not empty, make $G_c = G_c \cup \{C_c\}$; goto step 13.
(18) Set $Q_f$ = FAIL and make $G_c = G_c \cup \{C_c\}$ .
(19) If $G_c$ is not empty, set $C_c$ = first($G_c$), $G_c$ = rest($G_c$) and $I_c = I_{C_c}$.
(20) Goto step 5.
```

The class-based proof process is a generalisation of what is called 'object search' (e.g. in [19]). A class-based proof process eventually ends up to one or more instance-based processes. While an instance-based process goes bottom-up via inheritance links/relations, a class-based process goes top-down via specialisation links/relations, before it results in a number of instance-based processes. Also, while during an instance-based process there are successive extensions of the initial object's theory, during a class-based process there are successive restrictions of the initial context theory. This is illustrated in Fig. 4, where the route of a class-based proof process is indicated via the bold directed line. For the sake of simplicity, we assume that non-terminal classes have only subclasses and no instances. It is obvious that this theory restriction results in increasing efficiency.

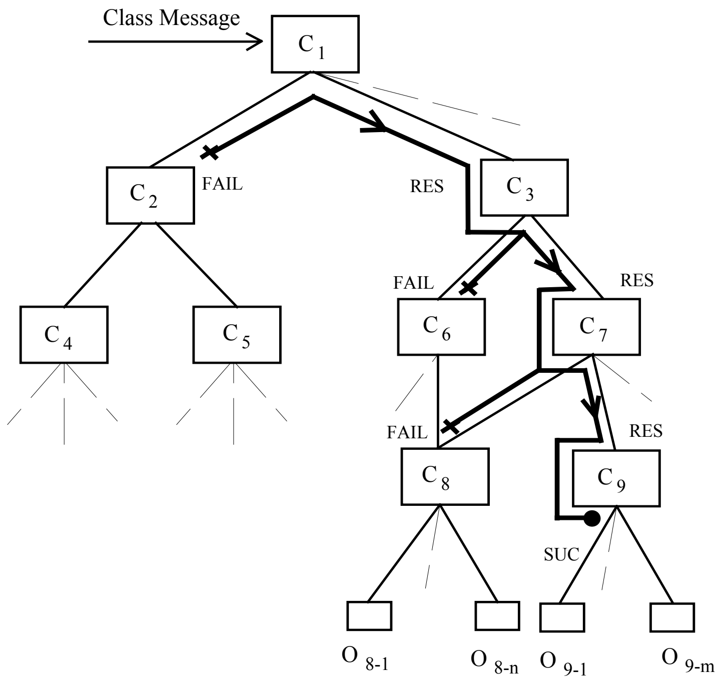

Fig. 4. Theory restriction in a class-based proof process.

6.2. Unification

The standard unification algorithm has been extended to be able to handle unifications related to typed terms as well as to procedure terms. The following rules, similar to those in a standard many-sorted unification algorithm, are used to deal with typed terms:

- a typed variable $v_i{:}C_1$ unifies with a constant O_i, if $\exists$ C_2, $O_i < C_2$ and $C_2 << C_1$.
- a typed variable $v_i{:}C_1$ unifies with a typed (skolem) constant/function $skc_i{:}C_2/(skf_i\ \ t_{i1} \ldots t_{in}){:}C_2$, if $C_2 << C_1$.
- a typed variable $v_i{:}C_1$ unifies with another typed variable $v_k{:}C_2$, if $\exists$ D_{12}, $D_{12} = C_1 \cap C_2$, where '$\cap$' means "intersection", that is $D_{12} << C_1$ and $D_{12} << C_2$, or $D_{12} < C_k$ and $D_{12} < C_m$ with $C_k << C_1$ and $C_m << C_2$. D_{12} is the greatest common descendant of C_1 and C_2 in the hierarchy. D_{12} is called then the *greatest lower intersection* (*gli*) of C_1 and C_2. (It corresponds to the greatest

lower bound, glb, in MSL [7]). D_{12} becomes the type of the unified variable, say v_m, that substitutes for the two variables, i.e. $v_m{:}D_{12}$. In case that gli is not unique, due to multiple inheritance, the most general (least specific) is chosen. This is determined by the rules established in Section 6.3.2.

To ensure soundness of derivations related to types (sorts) (see [24]), SILO does not allow for empty classes, that is classes having no instances.

For efficiency reasons, after unification of the predicates of two literals, unification of their object arguments is tried and then unification of their value arguments. A procedure term eventually becomes a constant. For efficiency and completeness reasons, its value is determined during unification of the associated literal. However, determination of its value may result in activation of a chain of resolution processes. This leads to a stronger interaction between unification and resolution than traditional.

When unification comes across a procedure term unification is suspended and it will wait until all of its arguments are instantiated to ground terms. This means that a series of resolution processes start off to infer values for the uninstantiated arguments of the procedure term. If instantiation of any argument fails, unification fails. Consequently, the corresponding function is executed. If execution fails, unification fails. Otherwise, its result substitutes for the procedure term and well-typedness of the atom is checked. If well-typedness fails, unification also fails.

The part of the unification algorithm for the evaluation of a procedure term is given below.

```
(1) If $t_{c2} = t_p$ , make $T_c = B_P$ and $S_c = S_c \cup T_c$ ; start $\mathcal{P}(S_c)$.
(2) If $\mathcal{P}(S_c)$ is successful, substitute variable bindings for $t_{p1}$ , ..., $t_{pm}$ ;
    execute $f^m$.
    (2.1) If $f^m$ execution is successful, substitute its result, $f(t_p)$, for $t_p$ and
          check well-typedness of $L_p$ .
          (2.1.1) If well-typedness checking fails, stop (failure).
          (2.1.2) If $t_{c1}$ unifies with $f(t_p)$, stop (success).
          (2.1.3) Stop (failure).
    (2.2) Stop (failure)
(3) Stop (failure).
```

The following terminology is used in the above algorithm. a_p is a procedure-axiom, L_p = **`first`**(a_p) is its procedure literal, since it is always is first in the axiom clause, and B_p = **`rest`**(a_p) is its body. L_p has the form $L_p \equiv (p^{n+1}\ t_1 \ldots t_p \ldots t_n\ t_o)$, where t_p is a procedure term of the form $t_p \equiv (f^m\ t_{p1} \ldots t_{pm})$. (For the sake of simplicity we assume only one procedure term in L_P). In a term pair from the two literals under consideration for unification, t_{c1} represents the term from the first literal and t_{c2} the term from the second literal.

6.3. Message based resolution

SILO uses as its basic resolution control strategy *linear resolution* (see e.g. [6]). However, the standard resolution process has been extended to mainly accommodate message passing. Because message passing is incorporated within the logical expressions, when a message expression is met during a resolution process, the process is suspended and another (sub)process starts off within another object according to the message instructions. Therefore, SILO's resolution process is called *message based resolution* (MB-Resolution). As usual, it aims at the production of the empty clause from a set of MPL clauses (axioms).

6.3.1. *MB-Resolution process*

In this section the MB-Resolution process is described. When two axioms are resolved and the resulting clause (resolvent) is not a message clause, then the resolvent is produced as usual. If not, an attempt is made to resolve the remaining literals in the contexts of their corresponding receivers, after substitution of the variable bindings produced by last resolution. This means that the current resolution attempt is suspended and a chain of new resolution attempts, one for each simple message starts. Bindings of a succeeded resolution attempt substitute for the corresponding variables, if any, in the remaining message clause before the next resolution attempt. Failure of a message literal to be resolved results in failure of the initial resolution and no resolvent is produced. If all message literals are resolved, then the empty clause is produced and a solution is found. Thus, one step of message-based resolution consists of a number of steps of ordinary resolution.

MB-Resolution process in a set of axioms S_c is more formally described by the following algorithm, where σ_c represents the current substitution (i.e. variable bindings), produced by the most recent successful resolution process, and *solution stack* is a set containing the solution already found.

```
(1) If there are no resolvable axioms in $S_c$
    (1.1) If solution stack is empty, stop (failure).
    (1.2) Stop (success).
(2) Select two resolvable axioms and compute their resolvent $r_{12}$ as usual.
(3) If $r_{12}$ is the empty clause, update solution stack; goto step 1.
(4) If $r_{12}$ is not a message clause, make $S_c = S_c \cup \{r_{12}\}$ ; goto step 1.
(5) Suspend current resolution process and organise $r_{12}$ as a sequence of
    simple messages: $r_{12} = ( M_1 \; M_2 \ldots M_n )$.
    (5.1) If $r_{12}$ is the empty clause, stop (success).
    (5.2) Set $M_c$ = **first**($r_{12}$), $R_c$ = **receiver**($M_c$) and $r_{12}$ = **rest**($r_{12}$);
          start $\mathcal{P}_{R_c}(M_c)$.
    (5.3) If $\mathcal{P}_{R_c}(M_c)$ is successful, check consequence retraction in $S_c$; then
          update the solution stack; make $r_{12} = r_{12} \circ \sigma_c$; goto step 5.1.
    (5.4) Goto step 1.
```

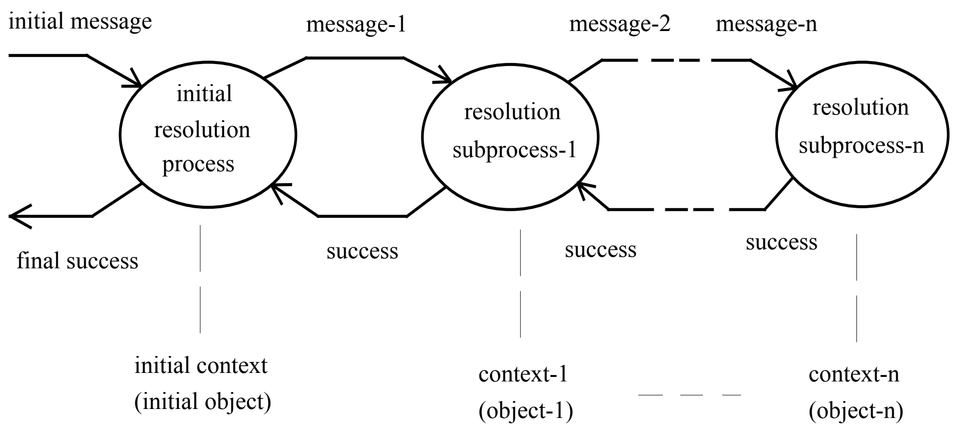

Fig. 5. Resolution with message passing.

An MB-Resolution in a set of axioms S is represented by $\vdash(S)$. The strong interaction between resolution and message passing is illustrated in Fig. 5.

6.3.2. *Control heuristics*

There are a few built-in resolution control heuristics that aim at increasing inference efficiency. The literals in any procedure-axiom are ordered in such a way that the procedure literal is always first, that is the head of the axiom, and is the only one to be tried for resolution. The rest literals constitute the body of the procedure literal and are used for instantiation of the variables in the head. The rest literals and the literals in any other axiom are ordered in such a way that message literals always follow all ordinary literals; positive ordinary literals precede negative ordinary literals.

The message literals in a compound message clause are ordered in such a way that a literal with a more specific receiver (object) is always before a literal with a less specific receiver. The rules for determining the specificity order of the receivers are the following:

(i) An instance is more specific than a class.
(ii) A class C_1 is more specific than a class C_2 if the distance of C_1 is greater than the distance of C_2.

The *distance* of a class is determined as the shortest path from the class to the object `object`. If C_1 and C_2 have the same distance, one is arbitrarily chosen.

7. Control Knowledge Representation

7.1. Meta-functions

SILO is a meta-level system, where object-level process control is achieved via (re)definition of certain functions, called *meta-functions* (see Section 2.4). SILO meta-functions are distinguished in deduction meta-functions and inheritance meta-functions. *Deduction meta-functions* concern deduction control. The deductive component of SILO's reasoner is a very restricted, in terms of control capabilities, and modified, as far as the reasoning process is concerned, version of ACT-P [12]. ACT-P is a resolution-based theorem prover, where resolution control can be specified by the user via a number of meta-functions. While ACT-P allows for changing the main resolution strategy, called the parent selection strategy, SILO has a fixed one, namely linear resolution. Thus, while ACT-P can be configured to incorporate strategies other than linear resolution, such as input resolution and P1-resolution, SILO's reasoner can be only configured to implement variations of linear resolution, such as SLD-resolution. This choice has been made for efficiency and pragmatic (deductions in SILO are less complicated than those in plain logic) reasons. So, deduction meta-functions of SILO are a small subset of ACT-P's meta-functions. Deduction meta-functions are stored in the 'deduction-control' component.

Inheritance meta-functions concern inheritance control; they are more fully described and justified in [22]. They control both complete inheritance and inheritance order. One can specify its own inheritance rules as far as complete inheritance is concerned; the arguments of the corresponding meta-functions are such that decisions based on the hardwired rules can be retracted. Also, because the query/theorem to be answered/proved is provided as an argument, problem specific rules can be also specified. The meta-functions concerning inheritance order specify the strategy to be followed for the inheritance path determination in cases of multiple paths. Inheritance meta-functions can have local or global effect, that is to specify local arrangements or general rules to be followed by all objects. Inheritance meta-functions are stored in the 'inheritance-control' component of an object. In the Appendix, the names of deduction and inheritance meta-functions are presented with a brief description of their effects.

7.2. Control defining language (CDL)

The integrated language composed of SDL and MPL is the object-level language of SILO; it concerns representation of domain knowledge in an object. The meta-level language of SILO is called *control defining language* (CDL) and concerns representation of control knowledge in an object. CDL expressions are actually the meta-function definitions. The control knowledge template in an object definition (see Section 4) consists of a number of CDL expressions:

```
<control-defs> ::= <ded-control-defs>
                   <inh-control-defs>

<ded-control-defs> ::= (<ded-function-def>*)

<inh-control-defs> ::= (<inh-function-def>*)

<ded-function-def> ::= (<ded-function-name> (<args>) <body>
```

<inh-function-def> ::= (<inh-function-name> (<args>) <body>

<ded-function-name> ::= deduction meta-function symbol

<inh-function-name> ::= inheritance meta-function symbol,

where <args> and <body> are defined as in a Lisp function definition.

To facilitate definition of the meta-functions, special built-in primitives, called *meta-primitives*, are provided that can access object internal information and perform a number of important actions. For a description of a number of them see [25].

8. Examples

In this section, some examples are presented that aim to illustrate some basic aspects of SILO. Specifically, in the first subsection a few SILO object definitions are presented; in the rest subsections examples that concern aspects of reasoning in SILO are presented.

Querying in SILO is enabled via the (built-in) primitive 'prove':

(prove <form>)

where <form> is an MPL formula. It returns either a set of variable bindings (a list of dotted pairs), if it is successful, or an indication for no solution, otherwise.

8.1. Object definitions

The following are object definitions in SILO. For the sake of simplicity, they are not complete in all of their parts.

```
(defclass human
          ;links
          ((mammal))
          ;attributes
          (((lives-in city) (1 1))
           ((sex (male female)) (1 1))
           ((income integerp) (1 1))
           ((son man)))
          ;definitional specs
            nil
          ;axioms
          ((num-of-legs 2 ?self)
           (eats ?x:meat ?self)
           (lives-in !works-in ?self)
           (allowance #(compute-allowance !income) ?self))
          ;procedures
          ((compute-allowance (income)
             (if (< income 1500)
                 (- 1500 income) 0)))
          ;deduction control
          ((select-l-literals (l-parent)
              (list (car l-parent)))
           (select-r-parent (r-parent)
              (remove-if-not #'positive-literal r-parent)))
          ;inheritance control
            nil)

(defclass man
          ;links
          ((human)(woman))
          ;attributes
          (((wife woman) (1 1)))
          ;definitional specs
          ((vess sex))
          ;axioms
          ((sex male ?self)
           (weight #(compute-weight !height) ?self)
           ((forall ?x)(=> (son !son ?x:woman)
                            (wife ?x ?self))))
          ;procedures
          ((compute-weight (height)
              (* 0.9 (- height 1))))
          ;deduction control
            nil
          ;inheritance control
            nil)

(definst  john
          ;links
          (man student)
          ;attributes
            nil
          ;definitional specs
            nil
          ;axioms
          ((lives-in paris john)
           (~ (likes swimming john))
           ((forall ?x)(=> (likes ?x !father)
                             (likes ?x john))))
          ;procedures
            nil

          ;deduction control
            nil
          ;inheritance control
          ((l-order-classes (sups theorem)
             (let ((preds (get-preds theorem)))
                  (if (member 'allowance preds)
                      (reverse sups)
                       sups)))))
```

8.2. Human relations

The object hierarchy used in this example is the following (where bold terms denote classes).

object	
human	
man	**woman**
john	carol
mike	mary

Also, the part of the knowledge base which is of interest for the example (only related axioms are presented in clausal form) is as follows:

human
((num-of-legs 2 ?self))

man	**woman**
((sex male ?self))	((sex female ?self))
((wife ?z:woman ?self)	
(~ (son ?v ?self)))	
(~ (son ?v ?x:woman))	

john	mary
((son mike john))	((son mike mary))
((num-of-legs 1 john))	

The attribute 'sex' has been declared as value essential.

Suppose that the following simple query is posed,

```
(prove '((exists ?x) (wife ?x john)))
```

that is *F* ≡ ((exists ?x) (wife ?x john)). Then *T* ≡ ((~ (wife ?x john))) is a simple message sent to `john`, as indicated by the fact that "john" is the object argument. It does not resolve with any axiom within `john`. Therefore, the axioms from `man` are directly inherited and the following set of clauses is set up:

(1) ((son mike john))
(2) ((sex male ?self)) ,
(3) ((wife ?z:woman ?self) (~ (son ?v ?self)) (~ (son ?v ?x:woman)))

with '?self ' bound to "john". The query clause ((~ (wife ?x john))) initially resolves with (3) and the following clause (resolvent) is produced:

((~ (son ?v john)) (~ (son ?v ?x:woman))).

This then resolves with (1), with ?v bound to "mike", and the following resolvent is to be produced,

((~ (son mike ?x:woman)).

However, because it is a message clause, the actual resolvent is not produced until the message literal resolves. The message literal (~ (son mike ?x:woman)) results in a series of messages being sent successively to each instance of the class `woman`, until a successful binding is found for ?x. So, (~ (son mike carol)) is sent to `carol`. However, since this query fails, the system then sends (~ (son mike mary)) to `mary`. Because this resolves within `mary`, success is returned to `john` and the suspended resolution process continues, producing the empty clause and returning **`((?x.mary))`**.

Suppose now the following query,

`(prove '((exists ?x) (& (sex female ?x:human) (son mike ?x))))`.

The clausal form of its negation gives, ((~ (sex female ?x:human)) (~ (son mike ?x:human))), which is sent to `human`. Because there is no resolvable essential axiom within `human`, it is sequentially sent to the subclasses of `human`, `man` and `woman`. There is no resolvable essential axiom in `man`, but there is in `woman`, from which ((~ (son mike ?x:woman))) is produced. Since `woman` is a terminal class, the produced message clause is sent to its instances one by one. It finally resolves with ((son mike mary)) in `mary`, the empty clause is produced and **`((?x.mary))`** is returned. Notice here, how top-down reasoning within classes reduces the number of instances to be searched, i.e. by excluding instances of `man`.

Finally, suppose the query,

`(prove '((exists ?x) (num-of-legs ?x mary)))`.

Then, ((~ (num-of-legs ?x mary))) is sent to `mary`, which eventually inherits ((num-of-legs 2 ?self)) from `human` with '?self ' bound to "mary", and the empty clause is produced with ?x bound to "2", that is **`((?x.2))`** is returned. However, for the query

`(prove '((exists ?x) (num-of-legs ?x john)))`

((~ (num-of-legs ?x john))) is sent to `john`, where it resolves with ((num-of-legs 1 john)), and the empty clause is produced with ?x bound to "1". That is, the more specific ((num-of-legs 1 john)) masks the more general ((num-of-legs 2 ?self)). This is a simple example of how default reasoning is performed in SILO.

8.3. Birds and snails

The domain knowledge used here is: "Birds and snails are animals and there are some of each of them. Also, there are some plants. Every animal likes to eat all plants or all animals much smaller than itself that like to eat some plants. Birds do not like to eat snails. Snails are much smaller than birds and like to eat some plants". (This is part of the well-known, in theorem proving, Schubert's steamroller problem [26]).

The object hierarchy of the domain is the following.

```
object
  animal                  plant
    bird     snail
     b1        c1              p1
```

where `b1`, `c1` , and `p1` represent arbitrary instances of `bird`, `snail`, and `plant` respectively. The object internal knowledge is as follows (presented in non-clausal form, for the sake of readability).

animal
```
(V ((forall ?x) (eats ?x:plant ?self))
   ((forall ?y) (=> (& (sm-th ?self ?y:animal)
                      ((exists ?z) (eats ?z:plant ?y)))
                    (eats ?y ?self))))
```

bird
```
((forall ?x) (~ (eats ?x:snail ?self)))
```

snail
```
((forall ?x) (sm-th ?x:bird ?self))
((exists ?x) (eats ?x:plant ?self))
```

There are no value essential attributes declared.

The query posed is,

`(prove '((exists ?y) ((exists ?x) (eats ?x ?y:bird))))`.

Thus, ((~ (eats ?x ?y:bird))) is sent to the class `bird`. Because there are no essential axioms in `bird` upwards, it is sent to its instance `b1`. Because there is no resolvable axiom, axioms from `bird` and `animal` are successively inherited so that the following set of clauses is created:

```
(1) ((~ (eats ?x ?y)))
(2) ((~ (eats ?s:snail ?self)))
(3) ((eats ?p1:plant ?self)
     (eats ?a:animal ?self)
     (~ (sm-th ?self ?a:animal))
     (~ (eats ?p2:plant ?a:animal)))
```

where ?y and ?self are bound to "b1". Now, (1) and (2) successively resolve with (3) with ?x restricted to "plant", and the following resolvent is produced ((~ (sm-th b1 ?u:snail)) (~ (eats ?p2:plant ?u:snail))), where ?u:snail is the unified variable produced from the unification of ?s:snail and ?a:animal, since `snail` is the gli of `snail` and `animal`. The resolvent is a simple message, thus it is sent to `snail` and the current resolution is suspended. Since there are no essential axioms in `snail` upwards, it is sent to `s1`, which inherits axioms from `snail` and a new resolution attempt starts from the following set of clauses

```
(4) ((~ (sm-th b1 ?u))
     (~ (eats ?p2:plant ?u)))
(5) ((sm-th ?b:bird ?self))
```

(6) ((eats (skf1 ?self):plant ?self))

where ?u and ?self are restricted to "snail". Now, (5) and (6) successively resolve with (4) and produce the empty clause. Thus, the initial resolution succeeds and the empty clause is produced, with ?x bound to "plant" and ?y bound to "b1", that is **`((?x.plant) (?y.b1))`** is returned. Notice that ?x is bound to a class name. This is an intentional way of expressing that a set of objects have a certain property. Taxlog [27] offers a similar capability.

8.4. Student affairs

In this last example, we demonstrate how procedural attachment is performed and the 'super' facility is used in SILO. We use the following hierarchy,

```
object
 human
   man        student
     john       john
     peter
```

where `john` is an instance of both `man` and `student`. We also use the following (partial) knowledge base.

```
human
; procedures
 ((compute-allowance (income)
      (if (< income 1500) (- 1500 income) 0)))

student
; axioms
 (((allowance #(compute-allowance ?inc) ?self)
    (~ (father ?x ?self)) (~ (income ?inc ?x))))
; procedures
 ((compute-allowance (income)
      (let ((v-income (* 0.75 income)))
          (* 0.5 (call-super 'compute-allowance v-income)))))

john
; axioms
 ((father peter john))

peter
; axioms
 ((income 1200 peter))
```

The query is "what allowance will John get?", expressed as

`(prove `((exists ?x) (allowance ?x john)))`.

Thus, (~ (allowance ?x john)) is sent to `john` and, after the inheritance of the axiom from `student`, the following set of clauses is set up,

```
(1) ((~ (allowance ?x john))
(2) ((allowance #(compute-allowance ?inc) ?self)
      (~ (father ?v ?self)) (~ (income ?inc ?v)))
(3) ((father peter john))
```

with ?self bound to "john".

First, an attempt is made to examine if clauses (1) and (2) are resolvable. Clause (2) is a procedure-axiom, which means that only the procedure literal participates in the process. Unification of the predicates of the literals of (1) and (2) succeeds. When unification comes across the computable term #(compute-allowance ?inc), it is suspended and the system tries to evaluate the argument of the term. To this end, the body of the procedure-axiom is joined to the above set of axioms as

```
(4) ((~ (father ?v ?self)) (~ (income ?inc ?v)))
```

and a resolution process starts. Clause (4) successively resolves with (3) and the clause (resolvent) ((~ (income ?inc peter)) is to be produced, with ?v bound to "peter". Because it is a simple message, it is sent to `peter`, where a resolution subprocess starts. ((~ (income ?inc peter)) resolves with ((income 1200 peter)) and the empty clause is produced with ?inc bound to "1200". The initial resolution then also succeeds. Thus, ?inc is instantiated and (compute-allowance 1200) is executed. During this execution, the homonymous procedure from `human` is called via the primitive 'call-super' in the body of 'compute-allowance' definition in `student`. Execution of the procedure finally returns '300', which substitutes for the procedure term. Now, ?x unifies with '300', hence (1) resolves with (2) and the empty clause is produced. Obviously, **`((?x . 300))`** is returned.

9. Related Work

Systems that in some way combine logic and objects can be distinguished in two broad categories. The first category includes hybrid knowledge representation systems that integrate notions from objects into logic. E.g. systems like KRYPTON [28] and KL-TWO [29], use the notion of the hierarchy of concepts in their terminological component, and a FOPC-based language in their assertional one. The unification algorithm is extended to take the information in the terminological component into account. Also, systems like [24], LOGIN [30] and Taxlog [27] employ a lattice/hierarchy of sorts alongside a FOPC-based component, and extend their unification algorithm to take into account sortal information, but in a different way from the former systems. SILO uses extensions similar to those in the latter systems. All these systems, although increase efficiency, suffer from two main drawbacks. First, they cannot represent exceptions and hence perform default reasoning. Second, they do not impose any actual structure on the domain knowledge, thus reducing naturalness. Their concepts (objects) are rather unstructured.

The second category includes systems that extend the logic programming paradigm to incorporate basic features from object-oriented programming, without affecting logic's basic constructs and mechanisms. Thus, unification algorithm remains unchanged and efficiency is not significantly improved. Also, their objects lack the structure of SILO objects. Furthermore, most of them face the problem from the programming point of view rather than that of knowledge representation. E.g. systems like [31], POL [32], SPOOL [18] and LAP [33] employ a standard class-based hierarchy model, which is not flexible enough for knowledge representation. Systems like MULTILOG [34], [19] and Plog [35] organise logical expressions in sets (objects) that communicate with each other via message passing in a way similar to that in SILO. However, except in Plog, objects are organised in a free way with no distinction between classes and instances. Also, systems like [36], [37], [38] and CPU [39] use the notion of inheritance hierarchy to (dynamically) organise sets of logical expressions with no communication between them, but they come with a sound semantics. CPU is the only system that uses a kind of meta-level knowledge to control inheritance. Most of the systems of this category have no means to express attribute declarations. However, due to their programming orientation, some of them support objects with a changing state during the course of computation (mutable objects), whereas SILO does not.

A common characteristic of the systems of both categories is that they give pre-eminence to logic, which is the basic reason of their drawbacks. There are, however, a few systems that use logic within an object-based framework, such as ORIENT84/K [40] and HSRL [41]. ORIENT84/K is a multi-paradigm system, that is the object-oriented part and the logic-based part in an object are distinct and there is an interface between them, via special variables. This way does not offer a unified representation, whereas SILO does. HSRL is a limited integration of frames and logic, by allowing Horn-type logical statements to be values of some slots, stored in meta-frames. This system does not exploit all the advantages that logic can offer, as SILO does. It takes a little from logic.

10. Conclusions

In this paper SILO, a general purpose hybrid knowledge representation system/language that integrates logic and objects, is presented. It is an extension of the work presented in [25, 42, 43, 44]. SILO gives pre-eminence to objects. Its object model comprise elements from both the class-based and the frame-based models. SILO uses separate representations for domain and control knowledge. An integrated language, that consists of two components, is used for the description of the domain knowledge in an object. The first component, called structure declaring language (SDL), concerns description structural knowledge in an object, by specifying its attributes and hierarchical links. The second component, a logic-based language, called message passing logic (MPL), is used for the description of factual and non-factual knowledge in an object concerning the values of its attributes. MPL is an extended form of FOPC that borrows elements from many-sorted logic. Thus, introduction of typed and evaluable increase expressiveness and conciseness of representation. MPL also allows procedural attachment, so that procedural representation is possible.

In SILO both the unification algorithm and the resolution procedure have been extended. The unification algorithm has been extended in a way similar to that in many-sorted logic, thus increasing efficiency of the deductions. It has also been extended to accommodate evaluation of computable terms, that may involve resolution attempts during unification. Resolution procedure has been extended to mainly incorporate message passing. The new procedure is called message-based resolution (MB-Resolution), where message passing plays a significant role. Sending a message to an object means sending a theorem to be proved in the context of the object.

From a logical point of view an object in SILO can be regarded as a theory. Under this consideration the inheritance of axioms can be seen as one way of expanding the theory. Thus, if a theorem cannot be proved from the local theory of an object, the local theory is extended by inheriting theories stored in its super-objects. The fact that some information may not be inherited from its super-objects, because it is not valid in the context of the current object, gives SILO the capability of default reasoning, as in all frame-based languages. Secondly, message passing through message literals can be seen as a way of temporarily changing the current theory (context) to that of the message recipient. Thus, apart from theory extensions, through inheritance, SILO allows for theory changes on demand, through message passing. So, both inheritance and message passing can be seen acting as a kind of control theory (meta theory) of the domain knowledge in the sense that they determine which propositions are used as axioms in a theorem proving request. This gives SILO, in many cases, an efficiency comparable to that of standard object-oriented programming languages.

Given the two types of a message, there are two modes of reasoning in SILO. In the first, as traditional, reasoning is performed in the context of an instance-object and is based on extensions of the initial theory via inheritance, thus going bottom-up in the hierarchy. In the second, reasoning is performed within a class-object using the essential axioms and is based on restrictions of the initial theory, rather than extensions, via specialisation, thus going top-down in the hierarchy. This reduces the number of instances to be searched.

SILO is a meta-level system. Its meta-level language, called control defining language (CDL), consists in a number of definitions of certain functions, called meta-functions, that can control deduction and inheritance processes.

There are, however, a number of missing characteristics from SILO, that may give rise to extensions. First, SILO does not offer direct constructs for supporting 'part-of ' relations or, in other words, composite objects. Also, it cannot do any classification, as e.g. systems with a terminological component can (essential axioms could be used as a basis for an extension to this end). Finally, SDL cannot express the fact that the type of a component value of an attribute is the conjunction or the disjunction of two or more other types (operators like AND and OR could be employed) as well as the range of its value.

A prototype of a basic core of SILO has been implemented in CommonLisp and tested via a number of examples, but there is still implementational work to be done. Since a potential problem of SILO might be the memory space required in cases of complicated deductions, where a number of long chains of messages exist, our future effort will be mainly focusing at implementing space efficient algorithms.

Acknowledgements

I would like to thank Han Reichgelt for useful discussions and Tony Cohn for some useful remarks on an earlier version of this work.

References

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

[2] H. Reichgelt, *Knowledge Representation: an AI perspective*, Ablex, NJ (1991).

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

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

[5] M. Stefik and D.G. Bobrow, *Object-oriented programming: themes and variations*, AI Magazine Winter 86 (1986) 40-62.

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

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

[8] K. Meinke and J. V. Tucker, eds, *Many-sorted Logic and its Applications*, John Wiley & Sons (1993).

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

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

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

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

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

[14] B. Nebel and K. von Luck, *Hybrid Reasoning in BACK*, in Z.W. Ras and L. Saitta, eds, Methodologies for Intelligent Systems **3**, North-Holland (1988) 260-269.

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

[16] R. W. Weyhrauch, *Prolegomena to a theory of mechanized formal reasoning*, AI **13** (1980) 133-170.

[17] K. L. Myers, *Universal Attachment: An integration method for logic hybrids*, Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), (1991) 405-416.

[18] K. Fukunaga and S. Hirose, *An Experience with a Prolog-based Object-oriented Language*, Proc. of the OOPSLA'86 Conference, (1986) 224-231.

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

[20] J. F. Allen and B. W. Miller, *The Rhetorical Knowledge Representation System: A User's Manual (for Rhet Version 1.4)*, Technical Report TR 238, University of Rochester (1988).

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

[22] I. Hatzilygeroudis and H. Reichgelt, *Handling inheritance in a system integrating logic in objects*, paper submitted to the Data & Knowledge Engineering Int. Journal (1996).

[23] P. Mello, *Inheritance as combination of Horn clause theories*, Inheritance Hierarchies in Knowledge Representation and Programming Languages, eds, M. Lenzerini, D. Nardi and M. Simi, John Wiley & Sons (1991) 275-289.

[24] C. Walther, *A Many Sorted Calculus Based on Resolution and Paramodulation*, : Pitman, 1987.

[25] I. Hatzilygeroudis, *Integrating Logic and Objects for Knowledge Representation and Reasoning*, PhD Thesis, University of Nottingham, UK (1992).

[26] M. E. Stickel, *Schubert's Steamroller Problem: Formulations and Solutions*, Journal of Automated Reasoning **2** (1986) 89-101.

[27] G. Montini, *Efficiency considerations on built-in taxonomic reasoning*, Proceedings of the 10th IJCAI, (1987) 68-75.

[28] R. J. Brachman, V. Pigman Gilbert and H. J. Levesque, *An Essential Hybrid Representation System: Knowledge and Symbol Level Accounts of KRYPTON*, Proceedings of the 9th IJCAI, (1985) 532-539.

[29] M. Vilain, *The Restricted Language Architecture of a Hybrid Representation System*, Proceedings of the 9th IJCAI, (1985) 547-551.

[30] H. Ait-Kaci and R. Nasr, *LOGIN: A Logic Programming Language with Built-in Inheritance*, Journal of Logic Programming **3** (1986) 185-215.

[31] C. Zaniolo, *Object-Oriented Programming in Prolog*, Proceedings of the 1984 International Symposium on Logic Programming, (1984) 265-270.

[32] H. Gallaire, *Merging Objects and Logic Programming: Relational Semantics*, Proceedings of the AAAI'86, (1986) 754-758.

[33] H. Iline and H. Kanoui, *Extending Logic Programming to Object Programming: The System LAP*, Proceedings of the 10th IJCAI, vol. I, (1987) 34-39.

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

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

[36] A. Borgi, E. Lamma and P. Mello, *Inheritance and Hypothetical Reasoning in Logic Programming*, Proceedings of the 9th ECAI, (1990) 105-110.

[37] E. Laenens, D. Sacca and D. Vermeir, *Extending Logic Programming*, Proceedings of the 1990 ACM SIGMOD Int. Conference on Management of Data, (1990) 181-193.

[38] L. Monteiro and A. Porto, *A Transformational View of Inheritance in Logic Programming*, Proceedings of the 7th ICLP, (1990) 481-494.

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

[40] M. Tokoro and Y. Ishikawa, *An Object-oriented Approach to Knowledge Systems*, Proceedings of the FGCS'84, (1984) 623-631.

[41] B. Allen and J. M. Wright, *Integrating logic programs and schemata*, Proceedings of the 8th IJCAI, (1983) 340-342.

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

[43] I. Hatzilygeroudis, *A framework for integrating logic and objects for knowledge representation and reasoning*, Proceedings of the 9th International Symposium on Computer and Information Sciences (ISCIS IX), Antalya, Turkey, (1994).

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

Appendix

Deduction Meta-functions

select-l-literals/select-r-literals:
Define the literals of the left/right parent clause to participate in the resolution process.

construct-resolvent:
Defines the way a resolvent is produced from its parent clauses.

combine-points:
Defines the general resolution space search strategy (blind or informed).

detect-infinite-path:
Defines strategies for detection of infinite paths in the resolution search space.

check-resolvent:
Defines strategies for pruning a branch in the resolution search space based on the
complexity of the literals or the clauses produced.

termination-condition:
Defines the conditions for terminating a resolution process (exhaustive, non-exhaustive search).

Inheritance Meta-functions

l-inherit-axioms:
Defines local inheritance rules, to satisfy local requirements.

g-inherit-axioms:
Defines global inheritance rules, to be followed by all objects.

l-order-classes:
Defines a local ordering strategy, to order the immediate (super)classes.

g-order-classes:
Defines a global ordering strategy, to change the default strategy.