

**Object-oriented Prolog:
design and implementation issues**

Jean-François Cloutier

**Department of Computer Science
McGill University, Montréal**

April 1989

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

© Jean-François Cloutier 1989

Abstract

In spite of its many qualities, Prolog is notably weak at handling state changes and at supporting modular programming. We designed SOAP as a clean, practical and reasonably efficient object-oriented extension to Prolog. We hoped that the strengths of object-oriented programming would compensate for Prolog's weaknesses.

We first discuss the issues involved in merging the logic and object-oriented paradigms, then we survey the different object-oriented Prologs and finally we present the design and implementation of SOAP which is entirely written in Prolog.

Résumé

Malgré toutes ses qualités, Prolog est reconnu comme étant faible dans la gestion des changements d'états et dans son support pour la programmation modulaire. Nous avons mis au point SOAP dans l'espoir qu'une extension objet à la fois propre, pratique et raisonnablement efficace comblera les manques de Prolog par les avantages de la programmation orientée-objet

Nous abordons d'abord les principes gouvernant la fusion des paradigmes logique et orienté-objet, puis nous effectuons un tour d'horizon des différents types de Prolog orientés-objet pour finalement présenter le design et l'implantation de SOAP qui est entièrement écrit en Prolog.

1. The logic and object-oriented paradigms.	1
a. What is a programming paradigm and why use it.	1
b. Complex problems and multi-paradigm languages.	3
c. Prolog: a practical logic programming language	5
(with some impracticalities)	
d. Object-oriented programming: strengths and	10
weaknesses	
2. Adding object-oriented programming capabilities to	15
Prolog: issues	
a. General principles governing the merging of	16
programming paradigms	
b. Prolog + objects: cancelling out weaknesses,	18
amplifying strengths	
3. A survey of Object-oriented Prologs.	20
a. Object-oriented concepts as additions to standard	20
Prolog	
b. Adding a Prolog class to an object-oriented language.	26
c. Prolog-based object-oriented languages.	31
d. Objects as perpetual processes in concurrent Prolog.	39
e. Extending unification to implement object-oriented	47
programming concepts	
4. SOAP: a practical object-oriented extension to Prolog.	53

a. The need for cleaner Prolog programs.	53
b. Requirements for a practical object-oriented extension.	56
c. An overview of SOAP.	59
d. The implementation of SOAP.	79
5. Conclusions.	85
a. An appraisal of SOAP and avenues for further improvement.	85
b. Object-oriented Prologs: how successful can they be?.	86
APPENDIX: the SOAP listings.	92
A. Creating, saving and restoring a SOAP image.	92
B. The first layer: message sending, utilities and pre-processor.	93
C. The second layer: SOAP's basic classes.	103

1. The logic and object-oriented programming paradigms

a. What is a programming paradigm and why use it

“A paradigm is a style of programming, supported by system facilities, that provides leverage in a range of programming tasks”[1]. The main purpose of a programming paradigm is to support a particular point of view taken by the programmer in his problem-solving activities.

Many different programming paradigms have evolved since the introduction of programming languages.

Functional programming uses the concept of function as its central theme; a program is a composition of functions, either primitive or composed, which takes in data and produces results after a series of transformations.

Logic programming describes the world in terms of clauses in first-order logic; a program is a set of axioms from which a theorem can be proved by deduction. Here the point of view is relational and theorem proving is the means of program execution.

Object-oriented (o-o) programming sees everything as being an object: each object is a self-contained unit since it alone can alter its state (or report on it). Objects get things done by sending each other messages. A message is a polite request to an object to report on its state or to alter itself.

Other programming paradigms exist (imperative, data-directed, constraint-based etc.), each one favoring a particular point of view and a particular structuring principle.

Each common programming language can, in the Turing machine sense, express whatever another can. Nonetheless, a language is said to support a given paradigm if it directly provides the primitive concepts of this paradigm (for example objects, inheritance etc in o-o programming.) and provides the means to write efficient programs using those concepts.[1]

Many factors will promote the choice of a paradigm for a given programming task. First is the familiarity of the programmer with the concepts involved and the cost of learning the requisite problem-solving skills, if needed. Then is the ease with which the perceived structure of the problem can be recreated using the concepts supported by the paradigm. For example, discrete simulation will benefit from a concurrent object-oriented treatment.

The cost of debugging and of maintenance is a factor of both the expressiveness of a language and of its ability to provide data abstraction. If a language is expressive and permits the writing of concise programs for a given problem, then a smaller amount of code has to be dealt with and its logic will be of a higher level; smaller, easier to understand programs are easier to maintain. Furthermore, if this language allows the repercussions of changes to be easily contained, then such modularity will greatly simplify maintenance.

b. Complex problems and multi-paradigm languages

The power of elision of a given paradigm (how much can be said in as few words as possible), as supported by a programming language, is a function of how well the concepts particular to this paradigm 'map' onto the problem at hand.

Given a simple problem, it is likely that one can find a paradigm which will be entirely satisfactory for the whole of it. However for larger, 'real-life' problems, the diversity of sub-problems makes it unlikely that a single paradigm will always be optimal. For example, the code needed to find a critical path in a project planning program can be very efficiently coded in Prolog. However when one starts implementing the user interface, one is forced to use very awkward idioms that stretch the capabilities of the language. It then becomes desirable to be able to use more than one programming paradigm concurrently.

The use of multiple paradigms presupposes that the programmer is capable of performing a 'paradigmatic decomposition' of his/her problem and that an appropriate multi-paradigm programming tool is available (one that supports more than one paradigm).

The programmer must also be aware of the particular set of concepts supported by a paradigm and should know in which situations they will be helpful and in which situations they will be either a hindrance or inadequate.

To be useful, a multi-paradigm language or environment must support powerful paradigms which strongly complement each other. By that we mean that each paradigm must be very expressive (the power of elision) and that each one's strengths must compensate to a large extent for the weaknesses of the other.

It is our purpose to demonstrate that logic programming and object-oriented programming are powerful paradigms that can be synergetically combined. One way of achieving the symbiosis of these two paradigms is to implement an object-oriented Prolog.

c. Prolog: a practical logic programming language (with some impracticalities)

Prolog[2; 3; 4; 5; 6] is a programming language founded on symbolic logic and developed[7] as a practical tool for logic programming[8]. Under the name Prolog, we group both standard Prolog as described in [2], its syntactic variants, and other Prologs such as Concurrent Prolog[6] which subsume standard Prolog. We hereafter assume that the reader is familiar with standard Prolog.

Prolog's outstanding features

Prolog's power lies in its use of unification and backtracking as its fundamental means to effect data manipulations and to control of execution.

Unification subsumes assignment, equality checking, pattern matching, structure composition and structure decomposition. Unification directly supports pattern-directed invocation and other powerful techniques such as incomplete data structures.

Prolog's use of backtracking in the execution of a proof, more appropriately depth-first search, makes it possible to better separate the logical aspects of a program from its control aspects[9]. As the behavior of a program is mostly inherent to the inference engine, programs with a declarative reading can be written. This declarative semantics, inherited from logic, is in addition to the procedural semantics of Prolog and makes for

more readable and easier to maintain programs.

Since clauses are both program and data, one can easily write interpreters in Prolog and in effect do meta-programming. When one uses the Definite Clause Grammar facility, Prolog becomes a singularly good tool with which to prototype and implement translators and compilers[10] making Prolog an excellent platform to implement new paradigms[1].

Logic variables are essentially assigned-once variables. They are declarative in the sense they do not have a history of taking on different values. Furthermore, as they are local to a clause and are the only kind of variables allowed, there is no fear of undesirable side-effects.¹ Prolog variables need not be instantiated to be operated on: two uninstantiated variables can be unified, meaning that they are constrained to eventually represent the same thing. In the course of execution, such equality constraints can accumulate leading to a technique known as constraint propagation.

Variable arguments of a procedure can assume a dual role; a variable argument will be used as 'input' if it is bound on invocation and used as 'output' if it is bound during execution of the procedure. Procedures can thus be multi-purpose. For example, the same append/3 procedure can be used either to concatenate two lists or to break a list in two parts.

Because unification of procedure arguments is the means by which information is shared and communicated, procedures can have multiple inputs and outputs. Prolog terms are recursively defined and can thus be of arbitrary complexity.

1. However, as will become apparent later, Prolog variables are not truly logical variables.

Through backtracking, a variable can take many different values: this is a high level form of iteration.² Such backtracking allows the writing of non-deterministic programs: these programs are said to contain ambiguities to be resolved later during execution. Such ambiguities are also called 'backtracking points' and support styles of programming dubbed 'don't care' and 'don't know' coding[4].

"The procedural semantics of a syntactically correct program is totally defined. It is impossible for an error condition to arise or for an undefined operation to be performed. This totally defined semantics ensures that programming errors do not result in bizarre program behavior or incomprehensible error messages"[5].

Prolog's weak points

Prolog's design was a series of compromises. Early theorem provers, which aimed at completeness, were hopelessly inefficient[11; 12]. By limiting oneself to Horn clauses (a clause with at most one unnegated literal)[8] and by superimposing a procedural interpretation to theorem proving using the linear input resolution strategy[11], one could obtain a practical logic programming language. The main sacrifice was lost of completeness.

This is not dramatic when one is mostly concerned with software engineering. What matters most then is that a programmer be able to solve a large range of problems efficiently using logic programming.

2. This can be shown to be both a blessing and a curse. More on that later.

In order to write practical programs, it is often necessary for the programmer to better control the otherwise default depth-first search algorithm used by Prolog. Prolog has thus introduced the cut, as a means to prune branches from the search tree, which has been likened to the infamous goto[13]. The cut, as a necessary evil, weakens or even inhibits the declarative reading of programs. Efficiency is not the only reason why a Prolog programmer must completely understand and use the procedural aspects of Prolog: failing to do so may lead to infinite searches.

At times, the programmer must explicitly use backtracking as a means of generating and recording alternate solutions or as a means of implementing state changes. This is done by inserting side-effects (printing to the screen, asserting to the database) which 'capture' the different values taken by a variable as a satisfactory value is searched. This defeats the logical nature of a Prolog variable which then behaves no more as an assigned-once variable.

All those programming 'tricks' undermine the benefits of logical programming. They are needed, nonetheless, because Prolog is singularly unsuitable at controlling computation and at supporting state changes.

Prolog is also weak at creating new data types. One can not create new data structures in Prolog other than those which can be represented by terms and/or lists.

Some significant constraints are imposed on Prolog data structures. Variables are constrained to be leaves in the tree described by a term; they can not express equality constraints anywhere else within the term[14]. Terms as record data structures are rather weak in that adding

another field to a term means modifying any reference to that term everywhere in the program (this underlines the need for data abstraction). Furthermore, a field is defined implicitly by its rank. One must always remember what the n th argument stands for.

Standard Prolog does not support concepts such as modularity, information hiding and data abstraction. Essentially, a Prolog program is one large, flat database of clauses. When one needs to effect state changes, one must rely on asserting to and retracting from the database which then behaves as a single global data structure. The introduction of modules in certain Prolog implementations serves only to fragment the database into smaller ones capable of information hiding.

Standard Prolog is essentially a sequential language which does not allow the exploitation of multi-processor architectures. Variants of Prolog have been developed to tackle the parallelism issue[6].

Finally, unification, being entirely syntactic, is ultimately restrictive. One would like, for example, 7 to be able to unify to *plus(3,4)* [15]. Also, AI programming is often interested in semantic unification (is an entity X a kind of Y ?) While one can use the logical implication in Prolog to implement a form of inheritance to serve the purpose of semantic unification, it has the disadvantage of lengthening proofs[14].

d. Object-oriented programming: strengths and weaknesses

“Many of the ideas behind object-oriented programming have roots going back to SIMULA[16; 17]. The first substantial interactive, display-based implementation was the SMALLTALK language[18]. The object-oriented style has often been advocated for simulation programs, systems programming, graphics, and AI programming. The history of ideas has some additional threads including work on message passing as in ACTORS[19], and multiple inheritance as in FLAVORS[20]. It is also related to a line of work in AI on the theory of frames[21] and their implementation in knowledge representation languages such as KRL[22], KEE[23], FRL[24] and UNITS[25].”[26]

Fundamental concepts of object-oriented programming

The object-oriented programming community is actively experimenting with new languages and new concepts. Many so-called object-oriented (o-o) languages have very little in common[26]. Efforts are currently underway to clarifying the issues and the concepts of o-o programming[27].

The core concept of o-o programming is, of course, the object. Objects are entities that combine both the properties of behavior and state since they contain both data (in the form of instance variables) and the operations (called methods) to manipulate that data. In other words, objects encapsulate the data structures they control[28].

"Objects may be contrasted with functions, which have no memory. Function values are completely determined by their arguments, being precisely the same for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience, its reaction to an operation being determined by its invocation history."[27]

An object-based language is one which supports objects.

The common properties (instance variables and methods) of a set of objects can be collected in a class. A class can be seen as a template from which object can be created in response to 'new' or 'create' operations. This object will know the methods and have the instance variables defined in its class. If the classes form a taxonomy, then the object also inherits from the superclasses of its class. If a class is allowed to have more than one direct superclass, then we have multiple inheritance. Otherwise, we have single inheritance. Inheritance is a form of resource sharing among classes[27].

Ambiguities occur when, for example, a method is defined both in a class and in its superclass. Such conflict is usually resolved by having the local method override the inherited one. In the case of multiple inheritance, a method could be inherited by two disjoint superclasses. There are many possible strategies for conflict resolution in multiple inheritance systems[29; 30].

An object-oriented programming language is one which supports objects, classes and inheritance.

An object activates another object by sending a message which carries information and the method to be invoked³. The receiver is solely responsible for deciding how to respond to a message. An object is assured of its integrity since only it can change its own state.

When an object does not know how to respond to a message, it can either cause an error condition or delegate responsibility to another object. Inheritance can be viewed as a special case of delegation.

The set of messages an object can respond to defines its protocol. This protocol describes the object's functional interface.

Object-based concurrency is attained by having objects execute concurrently.

The benefits of object-oriented programming

Most of the benefits of o-o programming can be traced back to two basic concepts: elision and data abstraction.

Class inheritance supports the technique of programming by specialization: this technique (also called differential programming[31]) reduces the amount of redundant information present in the system. Class inheritance allows the description of generic behavior at the appropriate level in the taxonomy, with the assurance that it will be inherited by every object that belong to any of the subclasses. In a nutshell, information can be added at the appropriate level of abstraction.

3. An object can also send itself a message.

Another form of elision is the ability to have objects of any type share the same protocol. For example, both points and numbers can be asked to double themselves etc. This means that a large system can contain a relatively small set of functional interfaces to its objects. This property is called polymorphism.

The fact that objects can only be dealt with through their functional interface is called data abstraction[28; 32]. It is a form of information hiding. Its main purpose is to contain the effects of program modification.

In the o-o context, if the implementation of an object's behavior is changed without altering its functional interface then no other object needs to know about it. Even the way an object records its state can be altered without side-effects, as long as the object still responds as expected to the same set of messages. O-o programs are extremely modular and can grow arbitrarily large without any increase in the cost of programming.

Object-based languages, such as ACT 1 or ACTORS[33], are very well suited for describing parallel processing. Since each object encapsulates both state and behavior and since objects communicate through messages, such languages are natural candidates for distributed, parallel implementations.

Objects of criticism

As a knowledge representation scheme, objects (and frames) are excellent at structural description. However, they are remarkably weak at representing incomplete knowledge. For example, assertions such as 'at least one of the valves is open or broken' is difficult to express in an object-oriented context.

Frame systems have unclear semantics. It is not clear whether a taxonomy contains information beyond structure definition. For example, the fact that class `ROCK` has three subclasses (`IGNEOUS_ROCK`, `SEDIMENTARY_ROCK` and `METAMORPHIC_ROCK`) does not necessarily mean that there are only three kinds of rocks. We might have `LARGE_GRAY_IGNEOUS_ROCK` as a combination of structural information (through multiple inheritance, for example)[34].

While inheritance has its benefits, it also has its problems: in large systems the complete description of a class is distributed over the taxonomy among the classes from which it inherits. It can be difficult to track and fully understand all the methods an object knows and their meanings, leading to a sort of 'spaghetti model' of inheritance[35].

2. Adding object-oriented programming capabilities to Prolog: issues

While one motivation behind adding o-o programming capabilities to Prolog is to compensate for as many of its weaknesses as possible, the ultimate goal remains to produce a whole greater than the sum of its parts.

This synergy effect can not happen if concepts from both paradigms are simply added together in a disjoint, complementary fashion. We want those concepts to inter-penetrate each other at a deep level, each one re-defining or amplifying the other. Non-determinism, for example, should help redefine the concept of inheritance; unification should augment the communication functionality of message passing etc. For such concepts to merge, they must be orthogonal to each other. Furthermore, the resulting collection of language features must be consistent, that is, these features must be able to co-exist.

Attempts at designing o-o Prologs differ both in their starting points (standard Prolog, Concurrent Prolog, Smalltalk, frame-based systems...) and in the depth to which the logic and o-o paradigms are integrated.

a. General principles governing the merging of programming paradigms

Merging language features require that they be orthogonal to each other.

“A collection of language features is orthogonal if no feature is a consequence of any of the other language features.”[27]

Proof of orthogonality for language features could (and probably should) be dealt with at a theoretical level, however an empirical approach can serve our purpose:

“A collection of features is orthogonal (independent) if, for every subset, there is a language that possesses that subset of features and no features in the complementary subset.”[27]

Since the set of o-o oriented concepts of object, class, inheritance and message exists in languages independently of the set of logic programming concepts (predicate, axiom, proof, logic variable, unification and non-determinism) and since they have been successfully combined in o-o Prologs, then both sets can be shown to be orthogonal.⁴

Integration of two paradigms is usually done by embedding a new paradigm into an environment which provides the other. One must then consider how much support is provided by an environment to facilitate such an integration.[1]

4. We are aware that this is demonstrating feasibility after the fact.

There must be appropriate facilities for building the primitives of the embedded paradigm (can unification be straightforwardly and efficiently implemented in an o-o environment?, can objects be easily implemented in Prolog? etc.). It must be possible to create new syntactic forms using existing structures in order to accommodate the representation of the added language features (for example, Prolog allows a certain freedom in creating infix, prefix and postfix operators which can then be combined with arbitrary complexity).

The embedding environment must be capable of providing the user with the ability to stay within the mindset of the paradigm he/she is using; for example, a user examining the execution of a program written in the embedded paradigm must not be presented with evidences of the underlying implementation.

Finally, the embedding environment must allow for an efficient implementation of the new language features.⁵

5. We will see that neither standard Prolog nor current o-o languages fully satisfy this requirement.

b. Prolog + objects: canceling out weaknesses, amplifying strengths

Canceling out weaknesses

Prolog's most glaring weakness, from a software engineering point of view, is its lack of modularity. Adding objects provides a fine-grained modularity to Prolog, finer than could be achieved through the use of modules.

An object provides the encapsulation of state information. Used within the context of Prolog, objects replace the flat database as the medium for recording state changes. State changes, which are side-effects within Prolog, can then be more easily handled with objects as it is effected more declaratively through their functional interfaces.

Objects are competent, self-referential data structures. Classes are user-defined types. Added to Prolog they provide greater flexibility in creating new data types and provide record-type structures with much greater flexibility than those implemented with Prolog terms (fields are now named and position independent, and one can add new fields to a structure without having to rewrite code using this structure).

Prolog's syntactic unification is augmented by inheritance which supports a form of semantic unification (any X is also a Y if X is a subclass of Y).

Objects are excellent at describing structural information, however, they are weak at representing incomplete information; any attempt to do so, beyond default values, is marred in unclear semantics.[34] Logic allows the representation of incomplete information in a straightforward manner and has clear semantics. Objects using logic to describe their respective states gain in their expressive capabilities.

Using logic to code an object's methods brings about the advantages of declarative programming. Furthermore, an object can easily be endowed with reasoning capabilities on top of its reactive capabilities.[36]

Amplifying respective strengths

Both logic and o-o paradigms have exceptional powers of elision.

In Prolog, unification can economically express all sorts of data manipulations and let variable arguments assume a dual role, and thus let procedures, assume multiple roles. Non-determinism supports the implicit expression of alternatives.

The class inheritance mechanism in o-o languages reduces the need for redundant information, and polymorphism reduces the amount of information the programmer must keep in mind as he/she interacts with objects.

Combining the powers of elision of both paradigms can yield a new, highly expressive language. This will become obvious as we survey different o-o Prologs.

3. A survey of Object-oriented Prologs

We have identified five major trends in the design and implementation of o-o Prologs:

- 1- Adding objects or frames as an add-on to standard Prolog
- 2- Implementing Prolog as a class in an o-o environment
- 3- Creating an o-o language on top of Prolog
- 4- Providing support for o-o programming clichés in a concurrent Prolog
- 5- Extending Prolog's unification mechanism to implement o-o concepts

a. O-o concepts as additions to standard Prolog

Standard Prolog is used as the implementation language onto which o-o concepts are added. Objects (or even frames) are meant to be no more than an extra data type (albeit a special one) available to the programmer. Such extensions are motivated by practical considerations: they are to compensate for some of Prolog's weaknesses (mostly lack of modularity and unsuitability at describing state changes). The intention here is not to rethink Prolog.

Such implementations are usually relatively simple, sometimes using unit clauses to store object descriptions. Often no distinction is made between instance variables and methods in how they are accessed and stored[37]. There is heavy reliance on the non-logical assert and retract which must

be used to change instance variable values or to record new instances and classes. In the worst cases, there is no provision for some sort of protection of an object's integrity; the Prolog database is wide open to any abuse. This lack of encapsulation tends to nullify the benefits of data abstraction which one associates with o-o programming.

There are three main solutions to the loss of data abstraction: the first one is to use the Prolog database in a way which prevents accidental misuse of objects, for example using only one procedure known (or unknown) to the user to describe all objects.⁶ The user could also be given special language constructs which he/she would be encouraged to use at all times when dealing with objects. Such constructs would support many of the expected clichés of o-o programming.[38]

The second solution is to create a new language on top of Prolog in which the user is forced to stay and which prevents any semantical misbehavior on his/her part. The third solution is not to use standard Prolog at all as the base language, but to use a Prolog in which representing an object state and state changes can be effected without side-effects to the database. Although this appears to be a contradiction, it is easily achieved in a concurrent Prolog[39]. As an alternative to the concurrency solution, a commercial Prolog (XILOG, Bull) supports property lists which can be used to store object descriptions[40]. Both of these options will be examined later on.

Implementations will differ, among other things, on which o-o programming concepts are implemented. While apparently all such o-o extended Prologs implement classes, not all of them implement metaclasses. Only

6. As we did with SOAP.

some will implement more 'exotic' concepts such as delegation and multiple inheritance.

Some experiments[41; 42; 43] have dealt with the addition to Prolog of frames, which are special kinds of objects. The concepts are essentially the same except for the vocabulary used (attached procedures for methods, slots for instance variables etc.) and the more complex behavior which each object (frame) initially possesses (if-added, if-needed etc. slots, default values, annotations etc.).

In order to get a more in-depth appreciation of this class of o-o Prologs, we now examine one of its representatives: ProTalk[38]⁷. We do not attempt to give a complete summary of this implementation. Instead we concentrate on its unusual features and those which, we believe, exemplify the problems generally associated with this brand of o-o Prologs.

ProTalk


In ProTalk[38], the database is used to store methods, instance variables and their values. No mystery is made on how methods are represented and invoked. Given the following message sent to an object:

send(+Object, +Selector, ?Message)⁸

the single method name used for the object is fetched (as any instance variable's value would be) using:

7. Since SOAP also belongs to this group, it is worthwhile to look at ProTALK in some details.

8. Prefixing a variable with + is a meta-notation which documents that it must be instantiated before invocation, - that it must be uninstantiated and ? that it can be either way.



fetch(method, +Object, -Method)

and then the call is made:

method(+Selector, +Object, ?Message).

send/2 allows messages to be sent with no information other than the selector.

As with normal Prolog procedures, backtracking will occur until a successful method is found. **send/2** and **send/3** are also backtrackable. An instance variable can have multiple values over which **fetch/3** will backtrack.

store/3 is **fetch/3**'s counterpart. It removes any previous associations which match the attribute's name and then uniquely associates a value with it:

store(?Attribute, +Object, +Value).

Attribute-value associations can be added with:

associate(+Attribute, +Object, +Value).

and retracted using

disassociate(?Attribute, +Object, ?Value).

This will retract all association which unify with `Attribute` and whose value is `Value`.

`disassociate/3` can have undesirable effects since it can be used to retract implicit attributes such as those giving the class of an object or its associated method name (these and other implicit attributes are implemented the same way explicit attributes are and are thus susceptible to abuse).

A ProTalk user can bypass the use of `send/2` or `send/3` in order to activate a method. However he/she is not told how instance variable-value pairs are implemented; a model is given

```
instance_name(Instance_variable, Value)
```

but the user is told not to rely on this model. Thus ProTalk attains a certain level of encapsulation through user ignorance.

Setting up a taxonomy in ProTALK is supported by the `new_class/2` and `new_subclass/3` procedures.

Adding a class as a subclass can be achieved by doing:

```
new_subclass(+Template, +Parent_s, +Method).
```

If `Parent_s` is a list then the new class will inherit all of the methods of every class in the list. ProTalk does not mention any strategy for conflict resolution.

It is possible, using the `new_class/2` predicate to create a new class with no ancestor. While this is not an error, it allows the creation of a set of classes which are not organized in a taxonomy. Thus in opposition to systems such as Smalltalk or SOAP where everything is ultimately an object, a ProTALK set of classes and objects can be totally unrelated. This, in our view, takes away the assurance of a conceptually cohesive object-oriented program and only retains the immediate practicality of defining classes as templates.

ProTalk provides services such as validity checking at instance creation time. Only one class is pre-defined in ProTalk to implement such system-wide generic behavior. There is no sense of a base system of classes and objects from which to borrow and to which to add in writing a program. Since o-o programming lends itself extremely well to the creation of large integrated libraries, the use of which greatly accelerates programming, it is surprising to find an o-o extension which would provide so little pre-defined functionality.

ProTalk supports a simple form of delegation as part of the generic behavior of an object. When a message is received and is not understood, a value for the delegate attribute of the receiver is sought and the message is then redirected to the object named by this value.

ProTalk is a somewhat minimal but useful o-o extension to Prolog. It is designed to combine with Quintus' ProWINDOWS⁹ object-oriented windowing package. ProTALK's greatest weakness is its lack of support for true data abstraction.

9. TMProWINDOWS is a trademark of Quintus Computer Systems, Inc.

b. Adding a Prolog class to an o-o language

Another way to implement an o-o Prolog is to start with an o-o language and then add a Prolog to the environment in the shape of a class . Prolog/V[44] and an unnamed prototype Prolog from Carleton University[45] have both been implemented on top of Smalltalk (respectively Smalltalk/V¹⁰ and Apple Smalltalk¹¹).

Prolog/V

Prolog/V is a structure copying implementation of a Prolog interpreter[46] written in Smalltalk/V. It implements most of the core of standard Prolog. It leaves out side-effect producing predicates (I/O) since such side-effects can be accomplished through calls to Smalltalk objects. Arithmetic functions are also left out for the same reasons. There are some minor lexical differences with standard Prolog such as a Prolog variable being a Smalltalk atom.

Prolog/V basically consists of a Prolog to Smalltalk/V compiler and of two classes which implement the functionality of Prolog: the Prolog and Logic classes. The Logic class implements the basic functionality of standard Prolog (unification, proof mechanism etc.). The Prolog class, a subclass of Logic, adds all of the built-in predicates.

Prolog clauses are defined in a class which must be a direct or indirect subclass of Prolog. Clauses with the same name are grouped together and

10. TM Smalltalk/V is a trademark of Digitalk Inc.

11. TM Apple Smalltalk is a trademark of Apple Computer Inc.

compiled as a single Smalltalk/V method. A class inherits all of the methods of its superclass, except when overridden, in the usual Smalltalk fashion. This means that a Prolog program can be decomposed into a taxonomy of classes. This feature provides the necessary support for modular programming which standard Prolog lacks.

Queries are messages sent to instances of Prolog subclasses. The answer to a query is an array of all possible instantiations of the free variables in the query. Prolog instances can thus communicate between each other. Since different classes can implement Prolog methods with the same names, Prolog/V can take advantage of the polymorphism feature of Smalltalk.

The interface between Prolog and Smalltalk is straightforward. Let's assume that the user has defined a Prolog subclass called Doctor which has some medical expertise. An instance of Doctor is generated from Smalltalk as follows:

```
aDoctor := Doctor new.
```

“We assume that aDoctor was declared as global”

We can now query the doctor from Smalltalk:

```
aDoctor :? cureFor(#baldness,aCure)12
```

Note that the :? is followed by a Prolog query.

12. #baldness is a constant and aCure is a Prolog variable.

Prolog might answer with:

```
((#aspirin #rest #callMeInTheMorning))
```

as all the possible values for the variable aCure.

Prolog clauses can access Smalltalk through the `is/2` predicate. The first argument is a Prolog term (variable or constant) and the second is a Smalltalk expression. On invocation, the Smalltalk expression is executed and its result is tentatively unified with the first argument. For example:

```
is(y, 2 * 6)
```

```
is(_, File pathName: 'data.dat' close)
```

Prolog/V is complemented with a LogicBrowser which allows the editing and viewing of Prolog programs in the style of the Smalltalk browser.

Debugging Prolog/V programs is problematic: there is no Prolog debugger. An error condition in Prolog/V invariably opens the Smalltalk debugger. One is then presented with the underlying Smalltalk implementation of Prolog/V; the illusion of working in the logic programming paradigm is not preserved at all times. It is also impossible to write efficient pure Prolog programs in Prolog/V¹³. Thus, Prolog/V can not be said to fully support the logic programming paradigm.

13. 47 LIPS (Logical Inferences per Second) in Smalltalk/V on an 8 MHz PC-AT

A Smalltalk-based Prolog with constraints

In [45] we are presented with a Smalltalk (structure sharing[47]) implementation which adds some extra features to those found in Prolog/V.

The first additional feature is the ability to give an object its own local facts and rules in addition to the ones it gets from belonging to a class. These local clauses are added by sending messages to an object whereas global clauses are defined using a browser. These local methods can only be invoked inside Prolog methods and are not directly accessible through a query sent to a Prolog object.

One difficulty we mentioned about standard Prolog was its inability to set up constraints other than equality on uninstantiated variables. One can not express that *X is or will have to be* smaller than *Y*, for example. Standard Prolog can only verify that constraints currently hold between ground terms but can not postpone and accumulate these constraints.¹⁴

This Prolog implementation of constraints took advantage of the o-o properties of Smalltalk. When a logical variable is constrained, a new kind of logic variable called a constrained logic variable is created which encodes the constraint.

“Since the implementation language is object-oriented, we have a different unifier for each significantly different class of objects (actually, only a handful). Each unifier (e.g. consider the unifier for Object) uses a three level priority scheme (constrained logic variable > logic variable > non-variable) to ensure that the receiver has the highest priority.

14. This feature is implemented as part of Colmerauer's PrologII.

Consequently, only the unifier for constrained logic variables needs to handle the complications of constraints [...] The priority scheme ensures that logic variables can be bound to constrained logic variables but never the other way around. Hence normal variables need never know about constrained logic variables.”[45]

As with Prolog/V, this augmented Prolog is written entirely in Smalltalk without kernel support. It is consequently unacceptably slow.

c. Prolog-based object-oriented languages

The types of o-o Prologs we have described so far have not brought the o-o concepts very deep into Prolog: such concepts have either been used to augment Prolog or they have been acquired because of the o-o nature of the implementation language.

More radical approaches are possible. One can rethink the fundamental concepts of o-o programming in relational terms. One can also decide to bring into standard Prolog a totally new dimension, such as concurrency, and do so from an o-o perspective. Whichever approach is taken, the end product is a language definition in its own right, rather than a mere extension of standard Prolog. This new language will usually subsume standard Prolog.

Redefining o-o concepts using relational semantics

“The question addressed here is whether the notions of inheritance and of procedural semantics attached to the object-oriented programming systems can be kept, or whether they have to be revisited in the light of a relational rather than functional paradigm. (...) This requires to adopt a success/failure semantics of backtrackable method calls, instead of the call/return classic mechanism. It also requires to allow for variable object calls and to introduce new types of methods dealing with non-monotonicity and determinism.”[48]

Gallaire's system POL distinguishes itself in its intention to fully merge the o-o concepts into the logical paradigm. Five requirements are listed which satisfaction would achieve this goal:

1- POL is to be a superset of Prolog

2- POL would support the usual o-o programming clichés: objects, classes, message passing, (multiple) inheritance.

3- Use of the relational framework to refine concepts such as inheritance and method evaluation: an analogy is here made between Prolog's search tree and the inheritance lattice as the meanings of a method can be distributed and searched bottom-up over the inheritance lattice.

4- Logic variables (and unification), non-determinism (and backtracking), the main features of logic programming, are to be applied consistently and without restriction to the o-o concepts; method calls are to be fully backtrackable (even over the inheritance lattice), one can send possibly uninstantiated messages to possibly uninstantiated objects. (Such completeness of evaluation not only requires careful implementation, but also careful control since it can lead to very expensive searches.)

5- Dynamic creation and destruction of objects.

Many o-o Prologs partially fulfill these requirements; ProTalk and SOAP, for example, will allow backtracking over a method call in a success/failure fashion instead of the call/return (commitment to first an-

swer) approach. However ProTALK and others do not complete the integration of logic and o-o concepts; uninstantiated objects will not be allowed to receive messages, local methods will override inherited methods, effectively preventing backtracking over the inheritance lattice.

The thoroughness with which requirements 3 and 4 are achieved is what sets apart this class of o-o Prologs from the others.

PROBE[40] follows the guidelines established in POL in attempting a maximally coherent and complete integration of the logic and o-o concepts. The coherence is achieved both in the o-o world (everything, including classes and metaclasses, is an object) and in the integration of both paradigms: multi-valued instance variables and multiply defined methods, value retrieval and method invocation through unification both backtrackable over the inheritance lattice (local clauses do not override inherited ones).

Controlling search over the inheritance lattice is imperative if efficient programs are to be written. In effect one needs a mechanism similar to the 'cut'. Whereas the cut prunes away branches from the Prolog search tree, this mechanism must disable search over certain arcs of the inheritance lattice. PROBE's 'cut_inheritance' primitive achieves this goal by cutting the choices on the last predicate making use of the inheritance lattice but does not affect the backtracking points of any other predicates.

Using this mechanism, PROBE implements deterministic and default methods which respectively, commit after first success and are invoked only if no other method was found after looking through the whole lattice. The other method type is non-deterministic. The type of a method is

given when it is defined.

As proposed in POL, the receiver of a message can be a free variable. This variable would then, through backtracking, assume the values of all the known objects. If the message is itself a free variable, it will also assume all method invocations understandable to the receiver. This goes well beyond the capabilities of most o-o Prologs which at most allow message arguments to be free variables.

SPOOL[36], like POL and PROBE, attempts an in-depth integration of logic and o-o concepts: it allows message receivers and method invocation to be free variables. Another dimension of integration it explicitly tackles is the closed-world assumption[49] under which Prolog programs are executed.

All o-o Prologs we surveyed exhibited some kind behavior based on this assumption. Completely integrating the closed-world assumption into the o-o context means that sending a message to an unknown object must fail but must not raise an error condition. The same can be said of an object receiving a message it does not understand. While SPOOL will, by default, raise an error condition in any of these events, it does provide constructs which allow the programmer to work consistently under the closed-world assumption: the programmer can use a message passing operator which relaxes the 'must-be-known' constraint on the receiver, the method invocator or on both.

Implementing concurrent o-o languages on top of standard Prolog

O-o programming offers excellent conceptual support for the expression of (distributed) parallel computations[50]. It provides a macro-level implementation model of concurrency (multiple, sequential Prolog processes executing in parallel) which does not get into the implementation difficulties of the micro-level approach (at the individual term level) of Parlog[51], Concurrent Prolog[6] or KL1[52] which are designed for execution on massively-parallel architectures.

In both SCOOP[53] and the CPU[54] (Communicating Prolog Units) model, objects have access to their own private Prolog databases. In both cases, an object is activated by sending it a message in effect asking it to launch a proof. Intermission[55], an actor-based language written on top of Prolog, also provides concurrency but in so doing, departs strongly from the Prolog language (very awkward syntax, very limited use of pattern matching, problems with backtracking etc.) and can not be said to be an o-o Prolog in a significant way.

A SCOOP program is a set of classes defined in an inheritance lattice (using a block structured syntax on top of the standard Prolog syntax) where each class describes a private database which each of its instances will, at first, possess. This database is split in two sections, static clauses and dynamic clauses. The dynamic clauses are restricted to facts and correspond to state description. The static clauses, which can be either facts or rules, correspond to methods. Conceptually, at object creation time, this private database is copied into the instance's initially empty database (only the dynamic clauses are actually copied). As their names imply, dynamic clauses can be asserted, retracted and replaced (retract

then assert) whereas static clauses can not¹⁵.

Program execution is a series of process activations which start with activation of a main process defined in a 'main class'. A process generates new processes through local and remote calls. A local call by an object accesses its own predicates; asserts and retracts of dynamic clauses are allowed. Another object's predicate is accessed through a remote call; remote asserts and retracts are disallowed since they would nullify the benefits of data abstraction.

A called predicate is first searched in the local database of an object, then in that of its class and then up the inheritance lattice until it is found. Consequently, locally defining a predicate overrides any inherited meanings it might have had.

Thus like most o-o Prologs and unlike POL or PROBE, SCOOP adheres to the view of inheritance as a defaulting mechanism. In order to have access to an inherited but overridden predicate, a 'super' special receiver construct must be supplied (and it is) which allows the invocation of a method in the context of the sender object but interpreted one class higher than its own.

SCOOP provides support for explicit process creation, interprocess communication and synchronization. These constructs and others implementing simulated time complete the set of necessary tools to do discrete-event simulation.

15. It is possible to parametrise objects at creation time by inserting dynamic clauses in their private databases.

While SCOOP directly supports the main o-o programming concepts, the CPU model offers a more fundamental set of capabilities on top of which such concepts can be implemented. In a sense, the CPU model is a pre-object-oriented language.

Unlike SCOOP where the concept of an instance is closer to the standard o-o clichés (an instance is created through a 'new' message sent to a class), a CPU's instance is said to be the local state of a demonstration activity (the state of the Prolog search tree) performed by a P-unit which is a separate Prolog program. A P-unit is assimilable to a class. Demonstrations can and do occur (conceptually) in parallel.

Communication policies are explicitly defined in terms of meta-P-units associated (through a 'connect' request) to the ordinary P-units; when a goal is to be demonstrated by an instance, this instance first performs a default communication with its connected meta-P-unit. This default communication is in fact a standard goal, `todemo(Sender,Receiver,Current-Unit,Goal,Result)`, to be demonstrated by the meta-P-unit. A meta-P-unit is itself a P-unit with possibly its own meta-P-unit etc.

The meta-P-unit's clauses are in fact meta-rules since they control how every messages are to be interpreted. Inheritance, delegation, filtering of messages etc. must be explicitly implemented in the form of meta-rules. With the CPU model one can experiment with different schemes of inter-object communication, and in effect implement different o-o languages. However, as it is, the CPU model is not an object-oriented Prolog but an object-based Prolog.

A general problem with Prolog-based o-o languages is their poor performance. As these o-o languages try, among other things, to offer better control over computation through o-o programming, they do suffer from Prolog's own weakness in this area. In other words, a language implemented on top of Prolog and offering a different model of computation is very unlikely to be efficient. These languages should be viewed more as experiments in o-o language design (this is certainly true of the CPU model) than as practical programming tools.

d. Objects as perpetual processes in concurrent Prolog

Concurrent Prolog in a nutshell

Concurrent Prologs[6; 51] are attempts at realizing the potential for parallelism which exists in logic programs. Standard Prolog successfully carried logic into logic programming by using a new kind of reading of definite clauses (a procedural or problem reduction reading[8]). Concurrent Prologs carry logic programming into the parallel framework by employing yet another reading of definite clauses: the behavioral reading.

“In the behavioral reading, a unit goal is analogous to a process, a conjunctive goal is analogous to a system of processes, and variables shared between goals function similarly to communication channels. A definite clause is read behaviorally: a process A can replace itself by the system of processes that contain B1 and B2 and ... and Bn. A process terminates by replacing itself with the empty system.”[39]

In Concurrent Prolog[6]¹⁶, unification goes beyond the assignment, equality checking, equality constraining, parameter passing, pattern matching etc. roles it assumes under the procedural reading. It now adds inter-process communication to its roles, since processes can share variables through unification.

Read-only variables (variables annotated by a suffix question mark) have been added to support process synchronization: a process will suspend if

16. From now on our discussion of concurrent Prologs will focus on Concurrent Prolog.

all of its reductions require the instantiation of a read-only variable. Guarded clauses have also been added in order to control process activation.

A Concurrent Prolog program is a finite set of guarded clauses which are universally quantified logical axioms.

$$A :- G_1, G_2, \dots, G_m \mid B_1, B_2, \dots, B_n. \text{ where } m, n \geq 0$$

“To reduce a process A using a clause $A_1 :- G \mid B$, unify A with A_1 , and, if successful, recursively reduce G to the empty system, and, if successful, commit to that clause, and, if successful, reduce A to B . The reduction of a process may suspend or fail during any of these steps... The computation of the guard system G suspends if any of the processes in it suspends, and fails if any of them fails.”[39]

Objects as perpetual processes

Concurrent Prolog supports the expression of objects based on Hewitt's Actor model[56]. An object is an active, perpetual¹⁷ process to which messages are sent (this is the only mode of interaction allowed). Such a process responds by performing actions on its internal state and sending messages to other objects. The state of the object-process is the current values of its unshared arguments. The object changes state by reducing itself to itself with new values for its unshared arguments. The class definition is the actual definition of a process (in contrast to the activation state of one of its instances created through a goal demonstration request).

17. A process which calls itself recursively.

The shared arguments play the role of communication channels and are carried over on self-reduction. If a communication channel is unified to an incomplete data structure, for example an incomplete list like [Message?!Other_messages_to_come], a message stream is established; when Message? (a read-only variable) is instantiated, the process is waken up, responds to the message then calls itself with Other_messages_to_come? as the perpetuated communication channel.

An example program taken from [6] should clarify the issues:

```
/* [clear | S] will unify with a message stream headed by the  
message 'clear'. */
```

```
counter([clear | S], State) :-
```

```
    counter(S?, 0). /* The process recreates itself with  
                    state 0 and waits for the remaining  
                    message stream to be instantiated.
```

```
*/
```

```
counter([up | S], State) :-
```

```
    plus(State, 1, NewState),  
    counter(S?, NewState).
```

```
counter([down | S], State) :-
```

```
    plus(NewState, 1, State),  
    counter(S?, NewState).
```

```
/* Retrieves the current state without changing it */
```

```
counter([show(State) | S], State) :-
```

counter(S?, State).

**/* The process terminates when the message stream is empty */
counter([], State).**

Counters could be used in the following way:

**?- terminal(TerminalStream), /* Message stream from
 the terminal */
use_counter(TerminalStream?, Counter1Stream),
counter(Counter1Stream?, 0). /* a counter with initial
 state 0 */**

**/* On goal unification, unifies the counter's message stream
head to show(Val), waits for more messages as it writes the
value (once obtained) on the screen. */
use_counter([show(Val)|Input],
 [show(Val)|Command]) :-
 use_counter(Input?, Command),
 wait_write(Val).**

**/* Simply passes any other message to the counter. */
use_counter([X | Input], [X | Command]) :-
 dif(X, show(Y)) | use_counter(Input?, Command).**

**/* Writes X to the screen as soon as it is instantiated. */
wait_write(X) :-
 wait(X) | write(X).**

Concurrent Prolog objects have no name. In order to keep track of existing objects, one must keep in a data structure the communication channels used to access them. Communication channels implemented by logic variables support many communication techniques. Since these communication channels can be unified, instantiating a set of unified channels allows the broadcasting of a message to a set of objects.

We have also seen how message streams can be implemented using incomplete data structures as communication channels. Since these streams are lists, it is possible for an object to peek ahead of the next message for possibly a further one and decide to react accordingly (for example, if a message is followed by one which undoes all of the effects of the first, both could be ignored). More general forms of message filtering can be achieved.

Using the properties of communication channels, one can implement different forms of default programming (for example inheritance or delegation).

The following code[57] sets up a class 'rectangular_area' of which 'frame' will be a subclass:

```
/* Clearing the area described by the parameters. */  
rectangular_area([clear | M], Parameters) :-  
    clear_primitive(Parameters) |  
    rectangular_area(M?, Parameters).
```

```
/* Answering the current values of the parameters. */  
rectangular_area([ask(Parameters) | M], Parameters) :-
```


rectangular_area(M?, Parameters).

/* A rectangular_area is created with the same parameters as a frame (it would have been possible to add new state information) and with a channel M1 which the frame will use to communicate with its superclass (to achieve inheritance). */

create_frame(M, Parameters) :-

/* The superclass process waits on M1 */

rectangular_area(M1?, Parameters),¹⁸

frame(M?, M1).

/* On receiving the draw message, the message ask(Parameters) is sent to the superclass (by unifying its communication channel). Then the contour is drawn (when Parameters gets instantiated by the superclass). */

frame([draw | M], [ask(Parameters) | M1]) :-

/* Executes and succeeds only when Parameters is instantiated */

draw_lines(Parameters) |

frame(M?, M1).

/* When asked to refresh, a frame asks its superclass to clear its area. */

frame([refresh | M], [clear | M1]) :-

frame([draw | M], M1).

/* If a message X is none that a frame understands, it is simply passed as is to its superclass. */

frame([X | M], [X | M1]) :-

18. A similar listing found in [39] contains a bug since M? is used instead of M1?

**dif(X, draw),
dif(X, refresh) !
frame(M?, M1).**

This scheme is very general and allows forms of delegation other than the special case of inheritance.

Concurrent Prolog supports programming techniques which subsume specialized constructs found in o-o languages and this is taken to be a sign of expressive power. Furthermore, objects and their state changes are achieved without the use of side-effects. Concurrent Prolog programs simulating side-effects are therefore more amenable to verification techniques than programs written in non-declarative languages. This opens new possibilities in the domain of program verification and generation in general and in the o-o context in particular.

Defining classes and inheritance in Concurrent Prolog is very error-prone ([39] contains a bug in one of its example programs) and the programming style is verbose and not at all obvious. This problem can be taken care of by writing a preprocessor which will provide a syntax directly supporting the usual o-o clichés. VULCAN[58] is a higher-level o-o language implemented in Concurrent Prolog which serves this purpose. Mandala[52], written on top of KL1 (another concurrent Prolog similar to Concurrent Prolog), provides the same kind of support for o-o programming.

Such o-o languages will have to wait for an efficient implementation of a concurrent Prolog on a massively parallel architecture to truly become useful. When efforts such as [59] are completely successful, the o-o lan-

guages implemented on top of concurrent Prologs will not only be very elegant and expressive merges of the two paradigms, but they will be extremely useful as well for both systems[60; 61] and applications programming.

e. Extending Prolog's unification to implement o-o programming concepts

Whether they were implemented on top of standard or concurrent Prolog, or on top of an o-o language, the o-o Prologs we have examined so far were achieved by either extending the base language with programming idioms, constructs or new data types, or by creating a self-contained language in its own right.

We now examine a different approach, one in which standard Prolog itself is altered in a fundamental way in order to achieve some of the benefits of o-o programming, and, as a consequence, becomes a better starting point to develop an o-o Prolog.

The focal point of this approach is the realization that unification only offers a weak and syntactic form of inheritance. A pattern can be seen as a generic representation of a class of grounded terms.

`special_list([X, [X|_] | _]).`

describes all lists with at least two elements where the second one is a list which head is identical to that of the overall list.

A grounded term would be an instance of the class of terms represented by the above pattern if it unifies with it. `special_list([1, [1,2],3])` would be such an instance and `special_list([[a],[a]])` would as well. `special_list([[X,Y],[X,Y],Z])` would be regarded as describing a sub-set (or subclass) of the set of terms described by the above pattern. Subclassing

can only be achieved this way syntactically.

If one wants to use standard Prolog to represent IS-A (inheritance) relationships between types of individual (a whale is a mammal), one can not use unification in a straightforward manner.

```
mammal(X) :- whale(x).  
whale(X) :- sperm_whale(X).
```

In fact one has to rely on a syllogistic formulation which, while semantically correct, imposes an extra and costly resolution step involving a state transition with context saving and variable binding.

What we are trying to simulate with such syllogisms is actually a form of semantic unification. One way to achieve this, without an extra inference step, is to augment the unification mechanism such that it will look for a user-defined equality assertion if syntactic unification fails. This approach is taken by Prolog-with-Equality[62], LOGIN[14] and UNIFORM[15].

Prolog-with-Equality, an extended Lisp-based Prolog, allows the creation of a type hierarchy or, if one prefers, a class hierarchy with the inheritance mechanism implemented using equality assertions¹⁹.

“The role of objects [is] played by terms; the role of messages by relations. The concept of ‘class’ has no formal analog in Prolog-with-Equality. The effect of class structuring is accomplished by the use of

19. Uniform takes essentially the same approach at augmenting unification, except that it does it in a way such that backtracking is no longer needed.

'equals' assertions. A subclass relationship is indicated by a single 'equals' assertion containing terms for the sub- and super-class. The patterns of variables between the two terms and the body of the assertion express the relationship between the two classes."^[62]

For example, the following program shows how a method (perimeter/2) can be inherited by equilateral_triangle/3 from regular_polygon/4.²⁰

```
equals(equilateral_triangle(X,Y,Length),
      regular_polygon(X,Y,3,Length) ).

perimeter(regular_polygon(X,Y,Nsides,Length),
          Perimeter) :-
    Perimeter is Nsides * Length.
```

If one now asks the following:

```
?- perimeter(equilateral_triangle(10,20,300), Perimeter).
```

the query will succeed with Perimeter bound to 900.

Prolog-with-Equality imposes restrictions on equality assertions needed to prevent circularity and thus infinite computations at unification. Another restriction is the fact that unification in Prolog-with-Equality is deterministic (as in standard Prolog, only one most general unifier is produced). This restriction prevents the implementation of multiple inheritance within unification. This would have been achieved, if allowed, by having multiple equality assertions where a given term appears as first ar-

20. We have used a standard Prolog syntax instead of Prolog-with-Equality's Lisp-like syntax.

gument.

Prolog-with-Equality is a better starting point to implement an o-o Prolog than standard Prolog, since it already possesses an efficient inheritance mechanism. Another 'elaboration' of Prolog called LOGIN[14] extends unification with user-defined equality assertions (called signatures). It goes further in providing an o-o flavor by replacing first-order terms by a more general form resembling record structures with tagged fields (some-what like frames).

μ - terms subsume standard Prolog terms and consist of:

- 1- A root symbol, the type constructor denoting a class of objects (its Prolog equivalent is the functor).
- 2- Attribute labels, which are record field tags associated with sub- μ - terms.
- 3- There can be coreference among sub-terms.

Here is an example of a μ -term with co-references:

```
person(id => name( first => string;
                    last => X : string);
father => person(id => name(last => X : string)))
```

This term describes any person to have 1- an id which is a name with a first part being a string and a last part being X, also a string, 2- a father which is a person which last part of the name is identical to that of the person. (A person's father must have the same last name as the person).

In standard Prolog, the arguments of a term are labeled by their respective positions. A standard Prolog term must always be written with all of its arguments in the correct order.

LOGIN labels the arguments of a term: they can appear in any order and some may even be left out if they are irrelevant in a given context. Adding a field to a LOGIN type does not force every reference to that type in a program to be altered, as is the case in standard Prolog when one alters the arity of a procedure. In LOGIN, the role of an argument is defined by its label, not its position in the predicate as in standard Prolog. Consequently, LOGIN offers more facilities for data abstraction than standard Prolog, and it does so in a manner reminiscent of standard o-o languages.

Unification between two μ -terms is achieved by calculating their greatest lower bound, that is, their most general common sub-type.

t1 is a sub-type of t2 if:

- 1- the root symbol of t1 is a sub-type of that of t2 (this is defined by the user as $t1 < t2$ in what is called a signature).
- 2- all attribute labels of t2 are also attribute labels of t1, and the associated μ -terms in t1 are sub-types of their corresponding μ -terms in t2.
- 3- all coreference constraints are satisfied

It can be shown that LOGIN subsumes standard Prolog by showing that Prolog terms are special cases of μ -terms and that LOGIN's unification mechanism encompasses that of standard Prolog.

LOGIN facilitates the definition of complex objects in a hierarchy and provides an efficiently implemented inheritance model. There is no distinction between classes and objects beyond the fact that generic terms can be assimilated to classes and grounded terms to instances. As with Prolog-with-Equality, messages would be relations (μ -terms intended at describing relationships between individuals, themselves represented as μ -terms).

Both LOGIN and Prolog-with-Equality should be viewed as pre-object-oriented Prologs, with LOGIN somewhat closer to the o-o paradigm.

4. SOAP: a practical object-oriented extension to Prolog

a. The need for cleaner Prolog programs

SOAP (a Simple Object-oriented Addition to Prolog) was designed and implemented as a matter of practical necessity. We were involved in the implementation of a large system (a knowledge-engineering environment) using SD-Prolog²¹, an IBM PC²² Prolog, and we had run into some difficulties.

The software being developed required a sophisticated user-interface. The support and utilities provided by the development language, while quite good, were insufficient for our purposes. As we developed the program, we realized that some rather problematic trends were emerging:

1- Most of the coding effort went into the user-interface and much of it seemed repetitive as the control structures for the interface elements shared a similar form.

2- It became difficult to manage and handle the state information associated with each user-interface object. The large amount of such information and the need to frequently modify it led to a very non-declarative style of programming. This was felt as highly undesirable.

21. TMSD-Prolog is a trademark of Systems Designers plc and is marketed under licence from Quintec Systems Ltd.

22. TMIBM PC is a trademark of IBM Corp.

3- Controlling the interactions with the user meant using a very procedural form of programming. We had to resort to programming idioms which defeat the benefits of logic programming (repeat-fail loops etc...) and a lot of information had to be carried as arguments in procedures which altered the state of the user interface. This meant a very heavy and unwieldy style of programming.

4- Lack of memory became a problem as the user-interface code became larger and larger. One could see that a different, more economical approach had to be taken.

5- The software needed to manipulate complex record-type data structures: creating, destroying and modifying them. Prolog was felt to be rather weak at representing complex structured data and at handling them in a natural way.

It became clear that the user-interface had to be designed and implemented using a radically different approach. Our experience with Smalltalk and SunView²³, an object-oriented user-interface sub-system, had shown us the benefits of separating the generic functionality of a user-interface from the scenario of interactions. This could be done by representing user-interface elements as objects which are responsible for recording their states and for reacting to requests from the application program. These objects also know which application procedures to trigger when informed that certain events occurred.

23. TMSunView is a trademark of Sun Microsystems, Inc.

U We thus decided to take the object-oriented route and develop an object-oriented user-interface sub-system since this would provide a clear model for the realization of state changes and the control of interactions (through message passing). Furthermore, we felt that inheritance and the fact that objects would encapsulate state information would both significantly reduce the size of the code needed to implement the user-interface support. We also hoped that we would see the same productivity gains we had experienced from the use of 'differential programming' while working in Smalltalk.

As a further motivation, object-oriented programming was felt to be the proper way to implement and manipulate large, complex data structures we needed for our knowledge representation scheme.

The first step was the design and implementation of an efficient o-o extension to Prolog: SOAP.

b. Requirements for a practical o-o extension

SOAP was designed and implemented using the following guidelines:

- 1- Efficiency: useful programs can be written.
- 2- Expressivity: a significant merge of both logic and o-o paradigms.
- 3- Coherency: a self-contained and self-describing system (as much as possible).
- 4- Flexibility: the user chooses the optimal mix of SOAP and Prolog, and can side-step SOAP's default checking mechanisms and thus exchange validity checking for better performance.
- 5- Portability: for obvious reasons.
- 6- Usability: flexible programming and debugging tools.
- 7- Large capacity: large programs possible through virtual memory.

SOAP is not an experiment in o-o language design. It is a practical tool designed to add the benefits of o-o programming to Prolog without an excessive penalty in run-time efficiency. This does not mean that the o-o layer was added in an ad hoc manner; we aimed at merging both paradigms in the most powerful, elegant and coherent way we could without unduly compromising efficiency.

Our experience with Smalltalk had taught us the benefits of a coherent and organized set of preexisting classes. Therefore, most of SOAP was defined in SOAP, with a small kernel written in Prolog. An effort was

made at a conceptually cohesive set of basic classes with everything being ultimately an object.

SOAP is written entirely in Prolog and in SOAP: it is easier that way and it makes SOAP more portable. Whatever part of SOAP is written in Prolog has been carefully tuned to minimize backtracking and costly structure manipulations.

Since Prolog code is faster than SOAP code, both SOAP and Prolog code can be freely intermixed for optimal results. Whenever safe constructs (which do some form of checking) are felt to impose too much of a burden, unsafe but faster constructs can be used instead.

The temptation to do 'gold-plating' was stoically resisted: no feature was added if its cost in program efficiency was not fully offset by its general usefulness.

Multiple inheritance was experimented with but was not needed enough to justify the added cost to message passing (because of the extra backtracking points). Concurrency was not even considered since it would have meant implementing a different model of execution on top of Prolog: an interesting but very costly proposition.

The scope of inheritance was defined in the standard o-o manner: local methods and instance variable values override inherited ones. The cost of managing uninhibited inheritance over the network would not be offset by the benefit of conceptual clarity. Anyway, a 'super' construct, providing the 'manual' override of method shadowing, was felt to be an acceptable, if less elegant, alternative.

The main o-o clichés (message passing, messages to self and super, cascading messages, instance creation, instance variable manipulation, dependents and broadcasting etc.) are expressed in SOAP either as special Prolog operators or as SOAP-defined capabilities. SOAP has a friendly surface syntax and an efficient base syntax (for fast method invocation). A preprocessor translates the first into the other.

A minimal programming environment is needed. Most of it is provided by SD-Prolog's incremental compiler accessible from within the text editor: SOAP's preprocessor can thus be used within the text editor. To this we have added a message recorder (to optionally keep a trace of all messages sent). Finally the user can save a SOAP image:

?- save_soap.

This command simply files the listing of all object descriptions in memory and compiles it into a format which can be later loaded in memory very rapidly.

Since SOAP uses the Prolog database to store object descriptions, we had to make sure that non-intentional corruption of the SOAP image could be prevented. This is achieved by using a single predicate (ivar/3) to encode every object description; it is assumed that the user knows about this. An added benefit of this approach is SD-Prolog's support for virtual memory on a predicate basis. Declaring ivar/3 as a virtual predicate essentially allows arbitrarily large SOAP images.

c. An overview of SOAP

SOAP's cosmology

There are two types of objects in SOAP: classes and instances. A class defines the set messages any of its instances can respond to (and how), and the instance variables (with value definition) used to describe its instances.

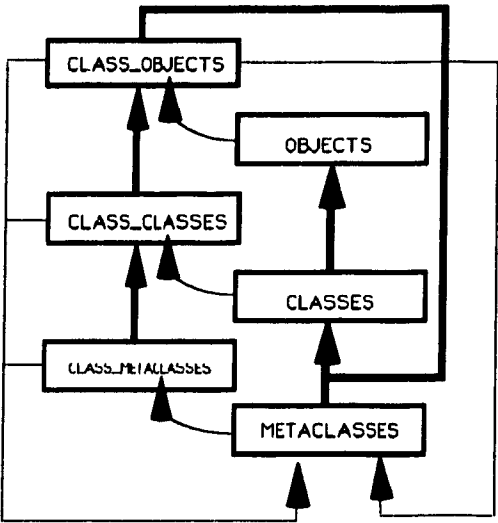
Each instance has a single class. Each class is an instance of a metaclass and has a single superclass (single inheritance model), except OBJECTS which has none.

The fundamental classes are OBJECTS and CLASSES (classes should but need not have plural names, except for metaclasses since they have only one instance).

CLASSES is a (direct) subclass of OBJECTS. All classes are direct or indirect subclasses of OBJECTS.

Every class X has its corresponding metaclass class_X of which it is the only instance.

Every metaclass is a direct instance of METACLASSES (even CLASS_METAClass, giving us a case of circularity). METACLASSES is a subclass of CLASSES.



(Thin lines show instance-of relationship while fat lines show subclass-of relationships.)

This organization corresponds very closely to that found in Smalltalk-80²⁴.

24. In Smalltalk-80, a class Behavior is used as an abstract superclass of both Class and Metaclass.

Instance variables

Every single piece of information about SOAP objects is recorded as instance variable values (taxonomic information, method definitions, instance variables domain definitions etc.) After all, in SOAP, everything is an object and an object's state is described by the values of its instance variables. Instance variables and their corresponding values are asserted in the database as:

`ivar(<object>,<ivar>,<a value>)`²⁵

For example:

`ivar(dogs,superclass,mammals).`

`ivar(fido,class,dogs).`

`ivar(dogs,instance,fido).`

`ivar(fido,age,12).`

An object can have more than one value for a given instance variable.

For example:

`ivar(maurice,diet,meow_mix).`

`ivar(maurice,diet,cat_chow).`

`ivar(maurice,diet,tender_morsels).`

25. Having the object's name as first argument takes advantage of SD-PROLOG's automatic indexing on first arguments. If no indexing were performed, SOAP would become very inefficient as more objects would be added. Ideally, the `ivar/3` procedure should also be indexed on the second argument.

Object names and instance variable names are Prolog atoms. Instance variable values can be any Prolog terms.

Creating a class

When a class is added to SOAP, one must give both its superclass and define the instance variables its instances will have (in addition to those inherited). One can optionally define class variables, which in SOAP are no more than the instance variables describing the class itself²⁶.

A class is created by using one of the following forms:

```
?- soap_class( name: <an atom>,  
               superclass: <a class name>,  
               instance_variables: [<ivar_definition>, ...]).
```

```
?- soap_class( name: <an atom>,  
               superclass: <a class name>,  
               instance_variables: [<ivar_definition>, ...],  
               class_variables: [<ivar_definition>, ...]).
```

The format of an instance variable definition is described below. SOAP allows the following shortcut:

<ivar_name>

26. SOAP has no such things as the class variables found in Smalltalk.

defaults to

```
<ivar_name> :: known(1,_)
```

For example:

```
?- soap_class( name: dispatchers,  
               superclass: objects,  
               instance_variables: [  
                                   screen,  
                                   status::known(1,[active,inactive])],  
               class_variables: []  
               ).
```

A class DISPATCHERS is created with superclass OBJECTS and instance variables SCREEN and STATUS. The definition of SCREEN defaults to

```
known(1,_)
```

which means that its only value must be known (thus can not be defaulted - more on that subject later-). The value of the instance variable STATUS must then be known and can either be ACTIVE or INACTIVE. Since no class variable is given, the other form soap_class/3, with no mention of class_variables, could have been used.

When a class is added to SOAP, any previous class of the same name and its metaclass are destroyed together with all of the class' previous instances. Then the new class is created and inserted in the taxonomy. Its

metaclass is automatically created and is itself inserted in the taxonomy. The class is made an instance of its metaclass. The instance variable definitions are checked for correctness (correct format and no redefinition of an inherited instance variable definition); class creation aborts and breaks the execution if an error is detected.

Instance variable definition format

When defining a class, the instance variables of its instances are described by being given a name, a maximum cardinality (maximum permissible number of different concurrent values) and a description of the domain of permissible values. The instance variable is also said to either allow default values (if a value is sought for a given instance and none exists, a default value defined in the class is obtained) or disallow default values (if a value is sought and none exists, the search fails).

Instance variable definitions (within class definitions) are of the form:

`<ivar_name> :: known(<cardinality>,<domain>)`

`<ivar_name> :: can_be(<cardinality>,<domain>)`

Disallowing default values (using `known/2` to define the instance variable's cardinality and domain) means that all of an instance variable's values are always explicitly known (asserted). Allowing default values (using `can_be/2`) means that either none or all of its values are explicitly given at any time. The set of an instance variable's default values are those which can be shown to belong to its domain.

When an instance variable values are said to be always known, a verifier is supplied as the domain descriptor; otherwise a generator is supplied.

A verifier is used only to check if a given value falls within the domain. A generator can act as a verifier but in addition it is also able to generate the domain.

A verifier is either a free variable, a list or a predicate with arity $n \geq 0$. A generator is either a list or a predicate with arity $n \geq 0$.

A free variable means that there is no restriction imposed on the domain.

For example, an instance variable *nickname* for all cats could be given an unrestricted domain (within the definition of class cats) as such:

```
nickname :: known(1,_)
```

A list is an explicit description of the domain. As a generator, it can be an incomplete list, like [blue, green, red | _], thus only providing a partial enumeration of the domain.

For example:

```
eye_color :: can_be(2 , [brown,green,blue | _ ]
```

When the domain description is a predicate, it is used to compose a message to be sent to the class of the instance which variable value is under scrutiny. The message is simply the predicate to which the value to validate or to obtain is added as first argument.

For example, given domain definition of the form

`<functor>(<arg>...),`

when domain membership verification or domain generation is required for the value of an instance's variable, the following message (with backtracking enabled -more on this later-) will be sent:

`<class_of_instance> *<- <functor>(Value,<arg>...)`

When the message is used in a verifying capacity, the first argument is instantiated and must be successful for the argument to represent a legal value (a value within the domain of permissible values). When the message is used in a generating capacity, it is uninstantiated at message sending time and must be instantiated during execution by a permissible value.

It is the responsibility of the class (or of one of its superclasses) to define a method which will answer messages requesting domain checking or generation. A fair number of such methods are defined in class OBJECTS. These methods can be used to describe a domain to be all instances of a given class or all positive integers etc.

For example:

```
/* any_given_number/1 */
ivar(objects,method, /* verifier */
    [
        [any_given_number_,Term],
        (
```

```

                                number(Term)
                                )
        }).

/* any_class/1 */
ivar(objects,method, /* generator */
    [
        [any_class_,Class],
        (
            classes *← [has_Instance,Class]
        )
    ]
}).

```

Note: the code shown above is in SOAP's base syntax where a method's code for a class is a list which is the value of the attribute METHOD for the given class (more on that later).

A class can redefine an inherited method which provides domain definition. This is the only mechanism allowed in SOAP which redefines an instance variable's domain; a class is otherwise not allowed to redefine an instance variable inherited from its superclass.

An instance variable's cardinality is the maximum number of different values it is permitted to have²⁷.

If it is a free variable, then any number of values is permitted. If it is an integer then at most that number of values is permitted. We will see later on how cardinality checking interacts with adding new values to an instance variable.

27. The fact an instance variable has a choice of different default values does not mean that it has all of these values: it has only one which can be any of them.

Accessing instance variables

Instance variable handling is always done using a special set of messages: `=<-`, `==<-`, `+<-`, `++<-`, `-<-`, and `-+<-`.

`<instance> =<- <ivar_name>(<value>)`

This message attempts to unify `<value>` to one of the values of the object's instance variable `<ivar_name>` (to check or retrieve a value if `<value>` is respectively bound or unbound). Both the receiver and the name of the instance variable must be instantiated when the message is sent.

If the object has no explicit value for this instance variable but is allowed to default it, then an attempt will be made to unify the argument with a default value.

It is backtrackable.

`<object> ==<- [<ivar_name>,<value>]`

This message differs from the above in that no unification to a default value is ever attempted. It is offered as a faster, 'no-frills' means to access instance variable values.

It is backtrackable.

`<object> -<- [<ivar_name>,<value>]`

If the receiver's instance variable has a value which unifies with <value>, then this value is retracted. It is not backtrackable; so must be considered as a side-effect producing message.

<object> +<- [<ivar_name>,<value>]

The value <value> is added as the last value of the object's instance variable. If the cardinality limit is exceeded, then the first of its values is removed. This gives a FIFO²⁸ behavior, and for instance variables of cardinality 1, it boils down to a simple replace operation.

If the value which is to be added can be unified to a known (non-default) one, the assertion does not take place and the call succeeds. The value is also checked for membership in the domain; if the check fails, an error message is sent and a break state is entered.

It is not backtrackable and thus causes side-effects.

<object> ++<- [<ivar>,<value>]

The value <value> is added as the last value of the instance variable. No cardinality or domain membership checks are performed. This is to be used only when speed is critical and when the user is willing to take the risk of adding an out-of-domain value or willing to risk 'overflowing' the instance variable. It is not backtrackable.

<object> -+<- [<ivar>,<value>]

28. First In First Out

is identical to:

```
<object> <- [ <ivar>, <value> ] ,  
<object> ++<- [ <ivar>, <value> ]
```

Method invocation

When a message that does not involve instance variable handling is sent to an object, an appropriate method is searched in the object's class and then up the hierarchy chain until one is found which will unify with it. As unification is used to select an appropriate method, arguments can be used in both input and output modes. More than one method can possibly be applicable. They can be successively invoked on backtracking.

A method can have more than one definition (or body) in a given class. If one definition is not successful (if its execution fails), the next one is tried.

Methods are fully backtrackable (they are Prolog programs optionally using message sending constructs). Local methods completely override inherited ones.

Message sending can be either deterministic or non-deterministic.

To send a backtrackable message:

```
<object_name> * <- <method_name>(<arg>, ...)
```

To send a deterministic message:

```
<object_name> <- <method_name>(<arg>, ...)
```

A deterministic message commits to the first successful method invocation.

Failure due to the absence of an appropriate method does not cause an error condition. However both the receiver and the method name must be instantiated at message sending time or else an error condition will be generated. An error condition causes a message to appear on the screen and puts the program in a break state (source code can be edited using SD-Prolog's editor and the execution can then be resumed or aborted).

Messages invoking methods can be cascaded. A cascade is a series of consecutive messages sent to the same object where the object needs only be named once.

```
objects <- <message> & <message> &...
```

In this particular case, because of the use of <-, the cascade as a whole is deterministic (not necessarily the individual messages themselves).

Method definition

A method is one of possibly many values of a class' instance variable called METHOD.

A method consists of a head and of alternate bodies. If the head unifies with the message, then a body is executed.

A method definition, in SOAP's surface syntax, has the following form:

```
?- soap_method(  name: <predicate>,
                  class: <an atom>,
                  meanings: [<body>, ...]).
```

The list of method bodies are alternate meanings which will be tried in succession on backtracking.

For example, the following method is defined with two bodies.²⁹

```
/* Take color of superview. If no superview or its color = none, take that of the screen */
?- soap_method(  name: eraser_color(Color),
                  class: views,
                  meanings: [
                      /* First body */
                      (self ==<- superview(S),
                       S ==<- background(Color1),
                       (Color1 = none,
                        frames ==<- current_screen(Screen),
                        Screen ==<- color(Color)
                       ;
                       Color = Color1
                      ) ),
                      /* second body */
                      (frames <- current_screen(Screen),
                       Screen ==<- color(Color) )
                      ] ).
```

The following method has only one body. The atoms 'self' and 'super' are special names and have the usual meanings: 'self' refers to the receiver of the message and 'super' also refers to the receiver but forces the corresponding method search to start from the superclass of the receiver's

29. This method is part of an unfinished user-interface toolkit written in SOAP.

class.

```
?- soap_method( name: react_to(Event),  
                  class: views,  
                  meanings: [  
                      ((self ==<- current_item(I),  
                        I <- react_to(Event)  
                        ;  
                        super <- react_to(Event)  
                        ) )  
                      ]).
```

A cut in a method body prevents backtracking within that body but will not prevent other bodies within the method or other methods to be tried. This is due to SOAP's use of meta-calls to effect method invocation. Because of the limited scope of the cut, SOAP enforces a more declarative style of programming than standard Prolog.

SOAP's programming environment

Very little effort has been spent on providing a complete development environment for SOAP. There is no browser, inspector or other necessities borrowed from Smalltalk.³⁰

SD-Prolog's integrated text editor and incremental compiler have supplied the necessary support for effective program development. In-situ compilation together with the necessary SOAP constructs to do class (re)creation, method (re)definition and message sending provided enough of a development environment to write large SOAP programs.

30. The development of such tools has to await the completion of the VIEWS object-oriented user-interface toolkit which itself prompted the development of SOAP.

What is sorely missed is a debugger which will abstract away the Prolog implementation of SOAP. SD-Prolog's debugger can not be used to follow a SOAP program since it does not enter a meta-call. Since method invocation is done by meta-calls, the first message will effectively create an execution 'black-out' in the debugger.

Short of writing a Prolog meta-interpreter in SD-Prolog, the easiest solution was to have SOAP keep a trace in a file of all messages sent (in chronological order with memory statistics) and then open a 'text view' on that file when the overall goal terminates.

The message recorder is invoked by doing:

```
/* A call can be a normal Prolog goal or a message sent to an
object. */
?- debug <call>, ... .
```

The syntax presented so far is a friendly, surface syntax. SOAP's 'real', base syntax is very similar but less friendly. In order to use the friendly syntax at the top level, one must use the operator 'do':

```
?- do <call>, ...
```

SOAP's pre-defined methods

Most of SOAP's functionality is defined in SOAP. All of SOAP's classes are defined using the base syntax (in ivar/3 format). Class and instance creation, for example, are implemented respectively by the methods `add_subclass/1` and `new/1`. The method `add_subclass/1` is defined in class `CLASSES` and `new/1` is defined in both `CLASS_OBJECTS` and `METACLASSES`³¹

Note that a method body, in the base syntax, is a list which first element is itself a list containing, in order, the name of the method, an argument (`Self`) which will be unified with the receiver when invoked, and the actual arguments of the method.

```
/* add_subclass/1 */
ivar(classes,method,
  [
    [add_subclass,Self,Name],
    (
      /* not an object already */
      not ivar(Name,_),
      /* give it a properly connected metaclass, thus making
         it an object */
      ivar(Self,class,Meta),
      Meta <- [new,Name],
      /* connect it */
      assert(ivar(Self,subclass,Name)),
      assert(ivar(Name,superclass,Self))
    )
  ]).

/* new/1 */
/* Given a class, create the corresponding metaclass and connect it as
   its subclass */
ivar(metaclasses,method,
  [
    [new,Self,Name],
    (
      concat('class_',Name,Meta1),
```

31. The reader is referred to the appendix for a complete listing of SOAP's classes and their associated methods in file `O_SOAP.PRO`.


```

        string_atom(Meta1,Meta),
        assert(ivar(Self,subclass,Meta)),
        assert(ivar(Meta,superclass,Self)),
        assert(ivar(Meta,instance,Name)),
        assert(ivar(Name,class,Meta)),
        assert(ivar(metaclasses,instance,Meta)),
        assert(ivar(Meta,class,metaclasses))
    )
}).

```

The necessary methods for instance variable manipulation (put_value/2, get_value/2 and get_value_no_default/2) are defined in OBJECTS.

```

/* put_value/2 */
ivar(objects,method,
    [
        [put_value,Self,lvar,Value],
        (
            (ivar(Self,class,C),
             C <- [in_domain,lvar,Value],
             !,
             A = ivar(Self,lvar,Value),
             /* add it if not subsumed by another already there */
             (call(A) ; assertz(A)),!,
             /* retract first value if now too many */
             (
                 Self <- [cardinality_violation,lvar],
                 retract(ivar(Self,lvar,_))
             ),
             true
        )
        ;
        Self <- [error,"Incorrect new value for an attribute!",
                 [Self,lvar,Value]]
    ],!
)
}).

```

```

/* cardinality_violation/1 */
ivar(objects,method,
    [
        [cardinality_violation,Self,lvar],
        (
            ivar(Self,class,Class),
            Class <- [cardinality,lvar,Max],
            nonvar(Max),
            A = ivar(Self,lvar,_),
            count_clauses(A,N),!,
            N > Max
        )
    ]
)

```

```

    )).

/* get_value/2 */
/* get a known value */
ivar(objects,method,
    [
        [get_value,Self,Ivar,Value],
        (
            ivar(Self,Ivar,Value)
        )
    ])

/* get a default one if permitted and none known */
ivar(objects,method,
    [
        [get_value,Self,Attr,Value],
        (
            not ivar(Self,Attr,_),
            ivar(Self,class,Class),
            Class *-< [default,Attr,Value]
        )
    ])

/* get a known value, no default */
ivar(objects,method,
    [
        [get_value_no_default,Self,Attr,Value],
        (
            ivar(Self,Attr,Value)
        )
    ])

```

OBJECTS provides the fundamental systems utilities: object printing, object destruction, dependency relationships à la Smalltalk with the associated signaling, broadcasting and updating capabilities, system information (can an object respond to a given message?), error handling, meta-message sending (send/1) etc³².

CLASSES provides the methods needed to add/destroy classes, add/retract methods and instance variable definitions to a class, remove an instance, handle the domain checking and the default value generation, enumerate instances etc.

32. The appendix is indexed and contains the complete program implementing SOAP.

METACLASSES does not do very much but it knows how to create a new metaclass and knows how a metaclass should self-destruct.

CLASS_OBJECTS knows how to find all of the instances of a class and how to remove them all. It also implements the `new/1` method which is used to create new instances of a class.

CLASS_CLASSES and **CLASS_METACLASSES** have no defined methods or instance variables. Right now they only bring in conceptual coherency to the class system, but should become more useful as SOAP evolves and as they get methods of their own.

d. The implementation of SOAP

A layered approach

SOAP is a three-layer system. The first layer is written in Prolog; it implements message passing, inheritance, the message recorder and SOAP's preprocessor (which converts SOAP's friendlier surface syntax into the base syntax). The second layer is the set of classes which define the basic system's capabilities of SOAP; they are implemented in SOAP's base syntax. The third layer is the application layer and is expected to be defined in SOAP's nicer surface syntax, with the help of the preprocessor.

Even though each layer seems to have its own particular language or dialect, it needs not always be the case. In fact, standard Prolog is used in both the second and third layers when it is felt to be the best thing to do. Furthermore, there is some message passing used in the kernel (to implement `=<-`):

```
/* unify ivar value message, default permitted; backtrackable */
=<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance =<- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,Class),
    ivar_of(Class,Ivar),
    !,
    Instance *<- [get_value,Ivar,Value].
=<-(Instance,Message) :-
    objects <- [error,"Incorrect attribute; can't unify(=<-)!",
                [Instance,Message]],
    !.
```

As a rule, in the third layer, one is expected to use only message passing in order to interact with objects. In the first layer and in the second layer to some extent, the Prolog database is directly modified as a means of changing objects states. This is done only when crucial optimizations are required and can not be achieved otherwise.

For example:

```
/* new_superclass/1 */
ivar(classes,method,
  [
    [new_superclass,Self,Super],
    (
      /* disconnects and reconnects itself */
      retract(ivar(Self,superclass,OldSuper)),
      retract(ivar(OldSuper,subclass,Self)),
      assert(ivar(Self,superclass,Super)),
      assert(ivar(Super,subclass,Self))
    )
  ]).
```

The first layer was designed to contain as little of SOAP's functionality as possible without overly compromising efficiency. The goal in designing the second layer was to have most of SOAP defined in SOAP. This objective led to some rather interesting code such as this self-describing statement part of CLASSES' definition:

```
ivar(classes,ivar,ivar::known(_,any_ivar_definition)).33
```

33. The reader should disregard the othermost ivar which is 'underneath' SOAP. Only the second and third one participate in the self-referentiality of the statement. This instance variable definition says that all classes can have instance variable definitions, and it describes their format. CLASSES is an instance of CLASS_CLASSES which is a subclass of CLASSES...

The base syntax

The base syntax is very close to the surface syntax. Essentially, messages are converted from the predicate form to a list form. For example,

```
an_object <- method(arg1,arg2)
```

becomes

```
an_object <- [method,arg1,Arg2]
```

in the base syntax.

When this message is interpreted by SOAP's kernel, a list is formed which is to be matched with the head of a method. In the base syntax, the head of a method has the following format:

```
[method_name, Self, <arg>, ...]
```

The first element of the list constructed from the message is the method name. The second is the receiver which will unify with Self and the rest are the message's arguments. This is achieved by the following code:

```
/* Methods execution is tried until success or failure */
/* fire_method/4 bypasses message passing for greater efficiency */
/* Single message */
fire_method(Instance,[Method|Args],Class) :- /* local */
    ivar(Class,method,[[Method,Instance|Args],Tail]), call(Tail). /* fire it */
/* inherited and not overridden */
fire_method(Instance,[Method|Args],Class) :-
    /* none found overriding */
    not ivar(Class,method,[[Method,_,_],_]),
    ivar(Class,superclass,Super),
    !,
```

```

        fire_method(Instance,[Method|Args],Super).
/* Cascade */
fire_method(Instance,&(Message,Messages),Class) :-
    fire_method(Instance,Message,Class),
    fire_method(Instance,Messages,Class).

```

The following is an example of a method definition in the base syntax, note the format of the message invocation in the method body:

```

ivar(class_objects,method,
    [
        [new,Self,Name],
        (
            Self <- [new_name,Name],
            /* put in new object (avoid message passing) */
            asserta(ivar(Name,class,Self)),
            asserta(ivar(Self,instance,Name))
        )
    ]
).

```

When a method is translated into the base syntax, all references to self and super are translated and all messages are converted to the basic SOAP format.

The special atom 'self' is transformed into a variable which is unified with the second argument (Self) of the method's head which is unified to the receiver at message passing time. This avoids the use of an instance variable called 'self' to implement self-referentiality.

Calls to 'super' are translated into special calls to Self. These special calls force the search of a method to start above class Class.

```

Self <- [<method>,<arg>...] @ Class
Self *<- [<method>,<arg>...] @ Class

```

The variable `Class` is instantiated to the class of the receiver at message sending time.³⁴

Efficiency and performance

SOAP's performance improved by a factor of 40 from its first implementation to its current state. It is felt to be reasonably performant when run on a fast micro-computer³⁵. Message passing is at least twice as slow as a standard Prolog procedure call (it takes longer when inheritance is involved). A single message can trigger other messages which meanings may have to be searched up the hierarchy. Consequently, response time to a message can degrade very rapidly as the taxonomy and object interactions grow in complexity.

For example, the following call, which looks for all objects in the system, takes half a second³⁶ to complete. It is assumed that no new classes or objects have been added to the basic SOAP system. The small preprocessing time is not taken into consideration:

```
?- do objects <- has_instance(I),fail.
```

This other call, which tests instance and class creation as well as destruction, takes slightly less than 2 seconds:

```
?- do objects <- add_subclass(rats),  
    rats <- new(N1) &
```

34. Refer to the predicator `soap_method/4`, `add_methods/4` and `soap_translate/4` in the appendix.

35. Anything faster than a 6MHz IBM AT™

36. All performance measurements were done on an 8MHz PC AT.


```
new(N2) &  
add_subclass(sewer_rats) &  
self_destruct,  
sewer_rats <- self_destruct.
```

Creating and destroying objects are expensive operations in SOAP. If the bulk of object and class creation is done as part of the development of an application and minimized during its execution, then acceptable performance can be achieved.

The domain checking construct `+<-`, which adds values to instance variables, seems to incur significant performance penalties because of the amount of message passing it triggers. We found ourselves using the risky, non-checking versions (`++<-` and `-+<-`) most of the time.

We used standard Prolog as one would use a lower level language to implement performance-critical code (drawing a box, filling a region, clipping a string etc.). Objects were then used as repositories for state descriptions to be loaded as parameters into fast Prolog procedures.

5. Conclusions

a. An appraisal of SOAP and avenues for further improvement

We found SOAP to be a very productive programming tool. It provided the benefits of o-o programming we had hoped for (modular program design, data abstraction, support for complex data structure representation and manipulation, default and differential programming) without taking away any of Prolog's best features (non-determinism, multi-purpose procedures/methods, pattern-directed invocation).

Since most of SOAP is written in SOAP, it became quite robust early in its own development and few bugs were found during the development of applications, such as an experimental user interface toolkit.

Even though the set of tools generally associated with an o-o programming environment are largely missing, productivity did not unduly suffer since SD-Prolog's own programming environment proved quite flexible. However, as the number of classes and methods grew, it became more and more difficult to navigate from one to the other.

A browser would be a very useful addition to SOAP's embryonic programming environment. A true debugger would be very useful. Nevertheless the post-mortem trace mechanism (the message recorder) was sufficient to rapidly detect most bugs. Finally, an inspector would certainly help in examining the changing states of the objects populating an application.

Certain improvements could be made to the syntax. An object should not always have to send messages to itself in order to examine or modify its own instance variables. It should be possible for an instance variable to receive messages which request state information or modification. The preprocessor would then translate in context these messages to the current form for efficient execution.

One should also be able to send messages to Prolog terms, for example `[1,2,3] <- reversed(L)`, so that it would be possible to write programs entirely in SOAP and still make use of Prolog's data types.

Porting SOAP onto faster Prologs would obviously speed it up. A Prolog allowing indexing on the second argument of the `ivar/3` procedure would certainly bring some improvement since this argument is used to name the instance variable (the first argument names the object).

Overall, we feel that SOAP is a useful and reasonably clean o-o extension to Prolog. However it must be mixed with standard Prolog for best results.

b.Object-oriented Prologs: how successful can they be?

The degree of success of an object-oriented Prolog design and implementation very much depends on what goals it set out to achieve. The aim might be to attain a better understanding of the design issues in integrating both logic and object-oriented paradigms, or it might be simply to provide a better system building tool.

Many object-oriented Prologs are object extensions to Prolog[37; 38] or Prologs implemented using object-oriented languages[44; 45]. The added paradigm inherits the characteristics of the implementation language paradigm in a rather straightforward way. Such implementations do not go very far in achieving a fundamental integration of both paradigms.

Our experience has shown that simple object-oriented extensions to Prolog, as with SOAP, produce a more efficient language, in terms of execution speed, than Prologs implemented on top of object-oriented languages. This is probably due to the fact these Prolog implementations were not optimized to any significant degree. Comparatively, adding object capabilities to Prolog is far simpler and does not necessarily incur a dramatic execution penalty.

Pushing basic object-oriented language capabilities, such as inheritance through semantic unification, into the core of Prolog[14; 15; 62], if optimized, could further improve the performance of object layers added on top of such 'enhanced' Prologs.

The object-oriented extensions to Prolog we surveyed provided a much finer integration of the two paradigms than the Prologs implemented on top of object-oriented languages. In the later case, a Prolog database would, in its entirety, be the set of 'logical' methods known to a class of objects.

More extensive experiments in object-oriented Prologs seem to be just that: experiments. While they provide powerful insights into merging the logic and object-oriented programming paradigms, they are much too inefficient to be used in building significant, deliverable programs. These

object-oriented Prologs[6; 15; 48; 51; 52; 53; 54; 55; 57; 58] are more than extended standard Prologs, they are new languages in their own rights, even though they often subsume standard Prolog.

POL and its derivatives[48; 40] are the object-oriented Prologs we surveyed which go the farthest in merging both paradigms: they allow concepts from both paradigms to interpenetrate each other, to become part of each other's definition. In POL, for example, indeterminacy and inheritance are interlocked.

In our opinion, the most promising, and satisfying, of these self-contained object-oriented Prologs is the Vulcan-Concurrent Prolog pair[6; [58]. Concurrent Prolog allows the creation of objects without necessitating side-effects and elegantly unlocks the inherent concurrency capabilities of objects. Vulcan provides a language preprocessor which makes it a lot easier to define classes and use objects than with Concurrent Prolog alone.

The expressive richness of the language is tremendous as it supports capabilities such as message passing (through communication channels), message queues (and message look-ahead), delegation (and thus inheritance) and more. All of these capabilities can be achieved without twisting the intents of the language, that is they can be implemented in a totally declarative manner.

When, and if, successful implementations of Concurrent Prolog on massively parallel computers become available, object-oriented Prolog may cease to be something of a curiosity to become an efficient and indispensable programming language.

- [1] Bobrow D.G. **If Prolog is the answer, What is the Question? or What it Takes to Support AI Programming Paradigms**, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11 (Nov. 1985)
- [2] Clocksin, W. & Mellish, C. **Programming in Prolog**, Springer-Verlag, Berlin, (1981)
- [3] Colmerauer, A. **PROLOG in 10 Figures**, IJCAI 83, 487-499
- [4] Sterling L. & Shapiro E., **The Art of Prolog**, MIT Press, (1986)
- [5] Pereira L., Coelho H. & Cotta J.C., **How to solve it with Prolog**, Ministério Da Habitação E Obras Publicas, Lisboa (1982)
- [6] Shapiro E. ed., **Concurrent Prolog, Collected Papers**, MIT Press (1987)
- [7] Roussel, P., **PROLOG - Manuel de Référence et d'Utilisation**, Groupe d'IA, UER Luminy, Univ. d'Aix - Marseille (1972)
- [8] Kowalski R., **Logic for Problem Solving**, North-Holland (1979)
- [9] Kowalski R., **Algorithm = Logic + Control**, Department of Computing and Control (1976)
- [10] Warren D., **Logic Programming and Compiler Writing**, DAI, Report no. 44, Univ. of Edinburgh
- [11] Nilsson N., **Principles of Artificial Intelligence**, Tioga Publishing (1980)
- [12] Chang C.L. & Lee R.C.T., **Symbolic Logic and Mechanical Theorem Proving**, Academic Press (1973)
- [13] Robinson J.A., **Logic Programming - Past, Present and Future**, New Generation Computing, vol. 1, no. 2 (1983)
- [14] Ait-Kaci H. & Nasr R., **LOGIN: a Logic Programming Language with Built-in Inheritance**, J. Logic Programming vol.3 185-215 (1986)
- [15] Kahn K., **Uniform -- a Language Based upon Unification which Unifies (much of) LISP, Prolog and ACT 1**, IJCAI-81, 933-939
- [16] Dahl O.J. & Nygaard K., **SIMULA - an algol-based simulation language**, Communications of the ACM, vol.9 671-678 (1966)
- [17] Papazoglou M.P., Georgiadis P.I. & Maritsas D.G., **The Programming Language Simula**, Computer Languages Vol. 9 No. 3/4, 107-131
- [18] Goldberg A. & Robson D., **Smalltalk-80: The Language and its Implementation**, Addison-Wesley (1983)
- [19] Lieberman H., **A Preview of ACT 1**, Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo no. 625 (1981)
- [20] Weinreb D. & Moon D., **LISP MACHINE MANUAL**, Symbolics Inc. (1981)
- [21] Minsky, M.A., **A Framework for Representing Knowledge**. In P. Winston (Ed.), **The Psychology of Computer Vision**, McGraw-Hill (1975)
- [22] Bobrow D.G. & Winograd T., **An Overview of KRL, a Knowledge Representation Language**, Cognitive Science, 1:1, 3-46 (1977)
- [23] Fikes R. & Kehler T., **The Role of Frame-based Representation in Reasoning**, Communications of the ACM, 28:9 904-920 (1985)
- [24] Goldstein I.P. & Roberts R.B., **NUDGE, a Knowledge-based Scheduling Program**, IJCAI-5, 257-263 (1977)
- [25] Stefik M., **An Examination of a Frame-structured Representation System**, IJCAI-79, 845-852

- [26] Stefik M. & Bobrow D., **Object-Oriented Programming: Themes and Variations**, AI Magazine, Vol. 6 No. 4 (1986)
- [27] Wegner P., **Dimensions of Object-Based Language Design**, OOPSLA '87 Proceedings., 168-182
- [28] Cohen T., **Data Abstraction, Data Encapsulation and Object-Oriented Programming**, Sigplan Notices, Vol. 19 No. 1, 31-35 (1984)
- [29] Borning A. & Ingalls D., **Multiple Inheritance in Smalltalk-80**, Proc. AAAI 1982, 234-237
- [30] Cardelli L., **The Semantics of Multiple Inheritance**, Proc. of the Conference on the Semantics of Datatypes, Springer-Verlag Lecture Notes in Computer Science, 51-66 (1984)
- [31] Rentsch T., **Object Oriented Programming**, Sigplan Vol. 17 No. 9, 51-57 (1982)
- [32] Snyder A., **Encapsulation and Inheritance in Object-Oriented Programming Languages**, OOPSLA '86, 38-45 (1986)
- [33] Pugh J.R., **Actors - The Stage is Set**, SIGPLAN Notices, Vol. 19 No. 3, 61-65 (1984)
- [34] Brachman R.J., Fikes R.E. & Levesque H., **KRYPTON: A Functional Approach to Knowledge Representation**, FLAIR Technical Report No. 16, Fairchild Lab. for A.I. (1983)
- [35] O'Shea T., **Panel: The Learnability of Object-oriented Programming Systems**, OOPSLA '86, 502-504 (1986)
- [36] Fukunaga K., Shin-ichi H., **An Experience with a Prolog-based Object-Oriented Language**, OOPSLA '86, 224-231 (1986)
- [37] Gullichsen E., **BiggerTalk: Object-oriented Prolog**, STP-125-85, MCC-STP, Austin, TX (1985)
- [38] Quintus Computer Systems, Inc., **ProTALK Programmer's Guide**, (1988)
- [39] Shapiro E. & Takeuchi A., **Object Oriented Programming in Concurrent Prolog**, New Generation Computing, Vol. 1, 25-48 (1983)
- [40] Gandilhon T., **Proposition d'une Extension Objet Minimale pour Prolog**, Séminaire de Programmation en Logique, Tréganel, 483-505 (1987)
- [41] Ito H. & Ueno H., **Zero: Frame + Prolog**, Proc. of the 4th Conference on Logic Programming, Lecture Notes in Computer Science, Springer-Verlag 78-90 (1985)
- [42] Amoux M., Becker G. & Thomas M.C., **Un Système de Frames Expertes en Prolog**, Séminaire de Programmation en Logique, Trégastel, 507-527 (1987)
- [43] Fornarino M. & Pinna A.M., **Intégration de Concepts de la Programmation Logique à un Langage de Schémas Paramétrés**, Séminaire de Programmation Logique, Trégastel, 143-170 (1988)
- [44] Digitalk Inc., **'The Prolog/V User Manual' (on-line documentation file accompanying Smalltalk/V)** (1986)
- [45] L. Londe W.R., **A Novel Rule Based Facility for Smalltalk**, ECOOP '87, 193-198
- [46] Mellish C.S., **An alternative to Structure Sharing in The Implementation of a Prolog Interpreter**, in Tarnlund S.-A., Clark K.L. (eds.), Logic Programming, Academic Press (1982)
- [47] Warren D., **Implementing Prolog - Compiling Predicate Logic Programs**, Dept. of Artificial Intelligence, Univ. of Edinburgh (1977)

- [48] Gallaire H., **Merging Objects and Logic Programming: Relational Semantics**, AAAI-86, 754-758 (1986)
- [49] Clark K.L., **Negation as Failure**, In *Logic Programming and Data Bases*, Gallaire & Minker (eds.), 293-324 (1977)
- [50] Yonezawa A. & Yokoro M (eds.) **Object-Oriented Concurrent Programming**, MIT Press (1987)
- [51] Clark K. & Gregory S., **Parlog: Parallel Programming in Logic**, ACM Trans. on Programming Languages and Systems, vol. 8 no. 1, 1-49 (1986)
- [52] Ohki M., Takeuchi A. & Fukawa K., **An Object-oriented Programming Language Based on the Parallel Logic Programming Language KL1**, Proc. of the Fourth International Conference on Logic Programming, 894-909 (1986)
- [53] Vaucher J., Lapalme G. & Malenfant J., **SCOOP, Structured Concurrent Object Oriented Prolog**, to appear in ECOOP '88
- [54] Mello P. & Antonio N., **Objects as Communicating Prolog Units**, ECOOP '87, 233-243, (1987)
- [55] Kahn K., **Intermission - Actors in Prolog**, in *Logic Programming*, eds. Clark K.L. & Tarnlund S-A, Academic Press, 213-228 (1982)
- [56] Hewitt C., **Viewing Control Structures as Patterns of Passing Messages**, Artificial Intelligence, vol. 8 (1977)
- [57] Takeuchi A. & Shapiro E., **Object Oriented Programming in Concurrent Prolog**, in *Concurrent Prolog*, collected papers, vol. 2, Shapiro E. (ed.), 251-273 (1987)
- [58] Kahn K., Tribble E., Miller M.S. & Bobrow D.G., **VULCAN: Logical Concurrent Objects**, in *Concurrent Prolog*, collected papers, Shapiro E. (ed.), 274-303 (1987)
- [59] Taylor S., Safra S. & Shapiro E., **A Parallel Implementation of Flat Concurrent Prolog**, *Journal of Parallel Programming*, Vol. 15 no. 3, 245-275 (1987)
- [60] Siverman W., Hirsch M., Houri A. & Shapiro E., **The Logix System User Manual Version 1.21**, in *Concurrent Prolog*, collected papers, vol. 2, Shapiro E. (ed.), 46-77 (1987)
- [61] Shapiro E., **Systems Programming in Concurrent Prolog**, in *Concurrent Prolog*, collected papers, vol. 2, Shapiro E. (ed.), 6-27 (1987)
- [62] Komfeld W.A., **Equality for Prolog**, IJCAI-83, 514-519 (1983)

APPENDIX: The SOAP listings

A. Creating, saving and restoring a SOAP image

/*FILE: MASTER.PRO

This file contains the code needed to create and restore a SOAP image.
***/**

clear_all :- clear_database(top_of_heap/0),assert(top_of_heap).

/* CONSTRUCTING A NEW SOAP IMAGE */

/*

/* If SOAP itself was modified, execute first these two lines. */

?- compile(soap).

?- load(soap).

**/* Saving all SOAP objects and all objects defined in SOAP by a hypothetical application
called soap_app into image.prm**

***/**

?- clear_all.

?- load(soap).

?- consult(o_soap).

?- consult(soap_app). /* replace "soap_app" by application name */

?- save_soap. /* Saves compiled ivar/3 in image.prm */

***/**

/* LOADING AN IMAGE */

?- clear_all.

?- load(soap).

?- load(image).

B. The first layer: message sending, utilities and pre-processor

```
/*
    FILE: SOAP.PRO

    SOAP: a Simple Object-oriented Addition to Prolog

    Author: Jean-François Cloutier

    Last modified: 28/04/1988

*/

/* IMPLEMENTATION NOTE:  STATE/1 is reserved by SOAP.
                        BAGOF/3 behaves like the standard FINDALL/3 */

/* Operators */
?- op(<-,xfx,1010). /* deterministic method invocation */
?- op(*<-,xfx,1010). /* non-deterministic method invocation */
?- op(=<-,xfx,1010). /* unify ivar value, default seeked if needed */
?- op(==<-,xfx,1010). /* unify ivar value, no default */
?- op(-<-,xfx,1010). /* "retract" ivar value */
?- op(+<-,xfx,1010). /* FIFO add/replace ivar value */
?- op(++<-,xfx,1010). /* fast_add ivar value, no cardinality check */
?- op(-+<-,xfx,1010). /* retract then fast_add ivar value */
?- op(@,xfx,1011). /* method search anchor: used to implement "super" */
?- op(:,xfx,900). /* ivar::value_definition */
?- op(debug,fx,1023). /* debugger */
?- op(do,fx,1023). /* soap top-level executive */
?- op(&,xfy,1009). /* cascade operator */

/* SOAP PRIMITIVES */

/* method execution */

/* Search method definition from superclass of Class and invoke */
@(*<-(Instance,Message),Class) :-
    debug_soap((Instance *<- Message @ Class)),
    atom(Instance),
    ivar(Class,superclass,Super),
    !,
    fire_method(Instance,Message,Super).
@(*<-(Instance,Message),Class) :-
    objects <- [error,"Illegal super message!",[Instance,Message]].
@(<-(Instance,Message),Class) :-
    debug_soap((Instance *<- Message @ Class)),
```

```

        atom(Instance),
        ivar(Class,superclass,Super),
        !,
        fire_method(Instance,Message,Super),
        !.
@(<-(Instance,Message),Class) :-
    objects <- [error,"Illegal super message!",[Instance,Message]].

/* non-deterministic method or cascade invocation */
*<-(Instance,Message) :-
    debug_soap((Instance *<- Message)),
    atom(Instance),
    !,
    ivar(Instance,class,Class),
    fire_method(Instance,Message,Class).
/* illegal message (no such instance or Message is var) */
*<-(Instance,Message) :-
    objects <- [error,"Illegal message!",[Instance,Message]].

/* deterministic method or cascade invocation.
Note that in the case of a cascade, it is the cascade which is deterministic, not the individual
messages. */
<-(Instance,Message) :-
    debug_soap((Instance <- Message)),
    atom(Instance),
    !,
    ivar(Instance,class,Class),
    fire_method(Instance,Message,Class),
    !.
/* illegal message (no such instance or Message is var) */
<-(Instance,Message) :-
    objects <- [error,"Illegal message!",[Instance,Message]].

/* attribute messages */

/* unify ivar value message, default permitted; backtrackable */
==<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance ==<- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,Class),
    ivar_of(Class,Ivar),
    !,
    Instance *<- [get_value,Ivar,Value].
==<-(Instance,Message) :-
    objects <- [error,"Incorrect attribute; can't unify(==<-)!",
                [Instance,Message]],
    !.

/* unify ivar value message, no default permitted; backtrackable */
==<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance ==<- [Ivar,Value])),
    atom(Instance),
    !,
    ivar(Instance,Ivar,Value).

```

```

==<-(Instance,Message) :-
    objects <- [error,"Incorrect attribute; can't unify (==<-)!",
                [Instance,Message]],!.

/* remove ivar value message; non-backtrackable */
-<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance -<- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,_),
    !,
    retract(ivar(Instance,Ivar,Value)),
    !.
-<-(Instance,Message) :-
    objects <- [error,"Illegal message; this instance does not exist!",
                Instance].

/* Adding/replacing by a new ivar's value */
+<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance +<- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,_),
    !,
    Instance <- [put_value,Ivar,Value].
+<-(Instance,Message) :-
    objects <- [error,"Illegal message; this instance does not exist!",
                Instance].

/* Adding without any consistency checking */
++<-(Instance,[Ivar,Value]) :-
    debug_soap((Instance ++<- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,_),
    !,
    assert(ivar(Instance,Ivar,Value)).
++<-(Instance,Message) :-
    objects <- [error,"Illegal message; this instance does not exist!",
                Instance].

/* fast replace ivar value message; non-backtrackable */
-><-(Instance,[Ivar,Value]) :-
    debug_soap((Instance -><- [Ivar,Value])),
    atom(Instance),
    ivar(Instance,class,_),
    !,
    (retract(ivar(Instance,Ivar,_)) ; true),
    assert(ivar(Instance,Ivar,Value)),!.
-><-(Instance,Message) :-
    objects <- [error,"Illegal message; this instance does not exist!",
                Instance].

/* Methods execution is tried until success or failure */
/* fire_method/4 bypasses message passing for greater efficiency */
/* Single message */

```

```

fire_method(Instance,[Method|Args],Class) :- /* local */
    ivar(Class,method,[[Method,Instance|Args],Tail]), call(Tail). /* fire it */
/* inherited and not overridden */
fire_method(Instance,[Method|Args],Class) :-
    /* none found overriding */
    not ivar(Class,method,[[Method,_,_|Args],_]),
    ivar(Class,superclass,Super),
    !,
    fire_method(Instance,[Method|Args],Super).
/* Cascade */
fire_method(Instance,&(Message,Messages),Class) :-
    fire_method(Instance,Message,Class),
    fire_method(Instance,Messages,Class).

/* Finds given Ivar as ivar of any instance of Class. Climbs up the hierarchy. Must not use
message passing or else the definition of '<-' would be circular.
*/
ivar_of(Class,Ivar) :-
    ivar(Class,Ivar,Ivar::_),!.
ivar_of(Class,Ivar) :-
    ivar(Class,superclass,Super),!,
    ivar_of(Super,Ivar),!.

/* Basic utilities */

a_number(1).
a_number(N) :-
    a_number(M),
    N is M + 1.

update(T) :-
    T =.. [F,Key,Val],
    T1 =.. [F,Key,_],
    (retract(T1) ; true),
    assert(T),!.
update(T) :-
    T =.. [F,Val],
    T1 =.. [F,_],
    (retract(T1) ; true),
    assert(T),!.

replace(Old,New,[Old|R],[New|R]) :- !.
replace(Old,New,[E|R],[E|R1]) :-
    replace(Old,New,R,R1).

deep_replace(Old,New,Item,New) :-
    Item == Old,!.
deep_replace(Old,New,Item,Item) :-
    (atomic(Item);var(Item)),!.
deep_replace(Old,New,[H|T],[H1|T1]) :-
    deep_replace(Old,New,H,H1),!,
    deep_replace(Old,New,T,T1).

```

```

deep_replace(Old,New,Pred,Pred1) :-
    Pred =.. [Func|Args],!,
    deep_replace(Old,New,[Func|Args],[Func1|Args1]),
    Pred1 =.. [Func1|Args1],!.

delete(EI,[E|R],R) :- !.
delete(EI,[H|R],[H|R1]) :-
    delete(EI,R,R1).

list_of_length(0,[]).
list_of_length(N,[_|R]) :-
    N > 0,
    N1 is N - 1,!,
    list_of_length(N1,R).

firstN(0,L,[]).
firstN(N,[],[]) :-
    N > 0.
firstN(N,[H|R],[H|R1]) :-
    N > 0,
    N1 is N - 1,!,
    firstN(N1,R,R1).

ends(L1,L2) :-
    reverse(L1,R1),
    reverse(L2,R2),
    begins(R1,R2).

begins([],L).
begins([H|R],[H|R1]) :-
    begins(R,R1).

/* Counting the number of clauses which match Clause in the database. The state 1 is used.
*/
count_clauses(Clause,N) :-
    state(1,0),
    break_clause(Clause,Head,Tail),
    count_clause1(Head,Tail),
    state(1,N),!.

count_clause1(Head,Tail) :-
    clause(Head,Tail),
    inc_state(1),
    fail.
count_clause1(_,_).

break_clause(Clause,Head,Tail) :-
    functor(Clause,_,2),
    Clause =.. [_|Head,Tail],!.
    break_clause(Clause,Clause,true).

inc_state(State) :-
    state(State,N),
    N1 is N + 1,

```

state(State,N1),!,

/*..... SOAP PRE-PROCESSOR*/

```
soap_class(name:N,superclass:S,instance_variables:I,class_variables:C) :-
    soap_class(name:N,superclass:S,instance_variables:I),
    writeln("Class variables:"),
    ivar(N,class,Meta),
    add_instance_variables(Meta,C),!.
```

/* The class Name replaces any object of the same name. (A bit crude, but will do for now.) */

```
soap_class(name:Name,superclass:Superclass,
            instance_variables:Instance_variables) :-
    (
        ivar(Name,_,_), /* if already a SOAP object, destroy it first! */
        writef("Destroying old %t...\n",Name),
        Name <- [self_destruct] /* if Name is a class, all instances are gone */
    );
    true
),!,
writef("Adding %t \nas kinds of %t...\n\n",Name,Superclass),
Superclass <- [add_subclass,Name],
writeln("Instance variables:"),
add_instance_variables(Name,Instance_variables),!.
soap_class(name:Name,superclass:Superclass,_) :-
    ivar(Name,superclass,Superclass),
    /* was inserted in hierarchy, so must be extracted */
    Name <- self_destruct,
    !,fail.
soap_class(name:Name,superclass:Superclass,_) :-
    objects <- [error,"Could not install class!",Name]].
```

add_instance_variables(_,[]).

add_instance_variables(Name,[Ivar::Def|R]) :-

```
    !,
    tab(4),writeln(Ivar),
    Name <- [add_ivar,Ivar::Def],!,
    add_instance_variables(Name,R).
```

add_instance_variables(Name,[Ivar|R]) :-

```
    tab(4),writeln(Ivar),
    Name <- [add_ivar,Ivar::known(1,_)],!,
    add_instance_variables(Name,R).
```

/* Replaces the old method by the new, if it exists */

```
soap_method(name:Message,class:Class,meanings:Meanings) :-
    writef("(Re)defining method\nfor %t\n\n %t",Class,Message),
    Message =.. [Name|Args],
    length(Args,L),
    list_of_length(L,Args1),
    (retract(ivar(Class,method,[[Name,_,Args1],_])),fail ; true),
    add_methods(Name,Args,Class,Meanings),!.
```

```
add_methods(_,_,_) :-
    nl,nl.
```

```

add_methods(Name,Args,Class,[Meaning|R]) :-
    write(" ").
    soap_translate(Meaning,Trans,Self,Class),
    assert(ivar(Class,method,[[Name,Self|Args],Trans])),!,
    add_methods(Name,Args,Class,R).

soap_translate((Conj,R),(Conj1,R1),Self,Class) :-
    translate_conj(Conj,Conj1,Self,Class),!,
    soap_translate(R,R1,Self,Class).
soap_translate(Conj,Conj1,Self,Class) :-
    translate_conj(Conj,Conj1,Self,Class).

/* Method translation */
translate_message(Message,Message,_) :-
    var(Message),!.
translate_message(&(First,Rest),Message1,Self) :-
    !,
    translate_message(First,First1,Self),
    translate_message(Rest,Rest1,Self),
    Message1 =.. [&,First1,Rest1].
translate_message(Message,Message1,Self) :-
    Message =.. [Method|Args],
    !,
    deep_replace(self,Self,Args,Args1),
    Message1 = [Method|Args1].
translate_message(Message,[Message],Self) :-
    atomic(Message).

translate_conj(Invoc,Invoc1,Self,Class) :-
    Invoc =.. [Arrow,Recipient,Message],
    soap_arrow(Arrow),
    !,
    translate_message(Message,Message1,Self),
    Invoc2 =.. [Arrow,Recipient1,Message1],
    (
        Recipient == self,
        Recipient1 = Self,
        Invoc1 = Invoc2
    ;
        Recipient == super,
        Recipient1 = Self,
        Invoc1 =.. ['@',Invoc2,Class]
    ;
        Recipient1 = Recipient,
        Invoc1 = Invoc2
    ),!.

/* arguments of a predicate can be messages (metacalling) */
translate_conj(Call,Call1,Self,Class) :-
    not list(Call),
    Call =.. [Func|Args],
    translate_all(Args,Args1,Self,Class),
    Call1 =.. [Func|Args1].
translate_conj(Call,Call1,Self,_) :-
    deep_replace(self,Self,Call,Call1).

```



```

translate_all([],[],_).
translate_all([Arg|Args],[Arg1|Args1],Self,Class) :-
    translate_conj(Arg,Arg1,Self,Class),!,
    translate_all(Args,Args1,Self,Class).

soap_arrow('<-').
soap_arrow('*<-').
soap_arrow('=<-').
soap_arrow('==<-').
soap_arrow('<+').
soap_arrow('<-').
soap_arrow('++<-').
soap_arrow('<+<-').

/***** TOP-LEVEL EXECUTIVE AND DEBUGGING COMMANDS *****/

/* Top-level executive */
do(Calls) :-
    retractall(debugging_soap/0),
    do_do(Calls).

do_do((Call,Calls)) :-
    do_do(Call),!,
    do_do(Calls).
do_do(Call) :-
    translate_conj(Call,Call1,_,_),!,
    Call1.

/* Top-level debugging */
debug(Calls) :-
    assert(debugging_soap),
    assert(debug_tab(1)),
    open(soap_debug,"soap.dbg",write),
    state(stdout,So,soap_debug),
    writeln("          SOAP TRACE"), writeln("%t:\n\n",Calls),
    stdout(So),
    (do_do(Calls);failure),!,
    stdout(So),
    close(soap_debug),
    view_debug_file("soap.dbg"),
    retractall(debug_tab/1),
    retractall(debugging_soap/0).

debug_soap(_) :-
    not debugging_soap,!.
debug_soap(Message) :-
    stdout(So),
    stdout(soap_debug),
    write_message(Message),
    write_stack_usage,
    stdout(So).

```

```

write_stack_usage :-
    statistics(stack,U1,S1),
    statistics(copy,U2,S2),
    statistics(trail,U3,S3),
    P1 is integer((U1 / S1) * 100),
    P2 is integer((U2 / S2) * 100),
    P3 is integer((U3 / S3) * 100),
    writef(" (S:%t,C:%t,T:%t)",P1,P2,P3),
    nl.

```

```

write_message(M) :-
    M =.. [ @,Invoc,Class],
    writef("Super[%t]: ",Class),!,
    write_message(Invoc).
write_message(M) :-
    M =.. [Arrow,Recipient,[Name]],
    !,
    writef("%t %t %t",Recipient,Arrow,Name).
write_message(M) :-
    M =.. [Arrow,Recipient,[Name|Args]],
    P =.. [Name|Args],
    writef("%t %t %t",Recipient,Arrow,P).

```

```

failure :-
    stdout(So),
    stdout(soap_debug),
    writeln("*** FAILURE! **\n"),
    stdout(So).

```

```

view_debug_file(File) :-
    stdio(Si,So),
    open(view_debug,3:8,19:64,b:yellow:b,_),
    stdio(view_debug),
    view(File,13),
    stdio(Si,So),
    close(view_debug),!.

```

/***** SAVING ALL SOAP OBJECTS *****/

```

save_soap :-
    open(save,'image.pro',write),
    state(stdout,So,save),
    nl,
    save_all_objects,
    stdout(So),
    close(save),
    compile('image.pro'),
    delete_file('image.pro').

```

```

save_all_objects :-
    ivar(Name,Ivar,Value),
    display(ivar(Name,Ivar,Value)),

```

```
        put(46),  
        nl,  
        fail.  
save_all_objects.
```

```
/* End of file */
```

C. The second layer: SOAP's basic classes

```
/* FILE:      O_SOAP.PRO */

/* DEFINITION OF SOAP'S BASIC CLASSES */

/* The core classes are defined as immediate assertions in Ivar/3 format */

/* OBJECTS

Ivars:          class,dependant,dependant_on

Methods:  changed/0,print_string/1,responds_to/2,
          should_not_implement/2,update/1,error/2,kind_of/1,
          self_destruct/0,broadcast/1,signal/2,
          instance_variable/1,put_value/2,get_value/2,
          get_value_no_default/2,
          add_dependant,remove_dependant,
          any_atom/1,any_class/1,any_value_definition/1
          any_kind_of/2,yourself/1,send/2, cascade/2

Class:          class_objects
Dependant:      none defined
Dependant_on:   none defined
Instance(s):    none defined
Superclass:     none defined
Subclass:       classes
*/

Ivar(objects,class,class_objects).
Ivar(objects,subclass,classes).

/* definitions for Ivars of instances of objects */
Ivar(objects,Ivar,class::known(1,any_class)).
Ivar(objects,Ivar,dependant::known(_,any_instance)).
Ivar(objects,Ivar,dependant_on::known(_,any_instance)).

/* method definitions */

/* yourself/1 */
Ivar(objects,method,
[
    [yourself,Self,Self],
    (
        true
    )
]).

/* put_value/2 */
Ivar(objects,method,
[
```

```

[put_value,Self,Ivar,Value],
(
  (Ivar(Self,class,C),
   C <- [in_domain,Ivar,Value],
   I,
   A = Ivar(Self,Ivar,Value),
   /* add it if not subsumed by another already there */
   (call(A) ; assertz(A)),I,
   /* retract first value if now too many */
   (
     Self <- [cardinality_violation,Ivar],
     retract(Ivar(Self,Ivar,_))
   )
  ;
  true
)
;
Self <- [error,"Incorrect new value for an attribute!",
        [Self,Ivar,Value]]
),!
)
)).

/* cardinality_violation/1 */
Ivar(objects,method,
[
  [cardinality_violation,Self,Ivar],
  (
    Ivar(Self,class,Class),
    Class <- [cardinality,Ivar,Max],
    nonvar(Max),
    A = Ivar(Self,Ivar,_),
    count_clauses(A,N),I,
    N > Max
  )
]).

/* get_value/2 */
/* get a known value */
Ivar(objects,method,
[
  [get_value,Self,Ivar,Value],
  (
    Ivar(Self,Ivar,Value)
  )
]).

/* get a default one if permitted and none known */
Ivar(objects,method,
[
  [get_value,Self,Attr,Value],
  (
    not Ivar(Self,Attr,_),
    Ivar(Self,class,Class),
    Class * <- [default,Attr,Value]
  )
])

```

```

    ).

/* get a known value, no default */
ivar(objects,method,
[
    [get_value_no_default,Self,Attr,Value],
    (
        ivar(Self,Attr,Value)
    )
]).

/* instance_variable/1 */
/* Name is an instance_variable of Self */
ivar(objects,method,
[
    [instance_variable,Self,Name],
    (
        Self <- [kind_of,Class],
        ivar(Class,ivar,Name::_)
    )
]).

/* changed/0 */
ivar(objects,method,
[
    [changed,Self],
    (
        forall(ivar(Self,dependant,Dep),
            Dep <- [update,Self])
    )
]).

/* update /0 */
ivar(objects,method,
[
    [update,_],
    (
        true
    )
]).

/* broadcast/1 */
ivar(objects,method,
[
    [broadcast,Self,Signal],
    (
        forall(ivar(Self,dependant,Deps),
            Dep <- [signal,Self,Signal])
    )
]).

/* signal/2 */
/* destruction signal received; remove dependency link */
ivar(objects,method,

```

```

[
    [signal,Self,Instance,destroyed],
    (
        Self <- [dependant_on,Instance]
    )
]).

/* do nothing: should be redefined by subclasses */
ivar(objects,method,
[
    [signal,_,_,Signal],
    (
        Signal \== destroyed
    )
]).

/* print_string/1 */
ivar(objects,method,
[
    [print_string,Self,Ps],
    (
        ivar(Self,class,C),
        string_atom(S,C),
        concat("an instance of ",S,Ps1),
        concat(" named ",Self,Ps2),
        concat(Ps1,Ps2,Ps)
    )
]).

/* responds_to/1 */
ivar(objects,method,
[
    [responds_to,Self,Method/N],
    (
        Self * <- [kind_of,Class],
        list_of_length(N,Args1),
        ivar(Class,method,[[Method,_,Args1],_])
    )
]).

/* should_not_implement/2 */
ivar(objects,method,
[
    [should_not_implement,Self,Method,Args],
    (
        string_atom(M,Method),
        ivar(Self,class,Class),
        string_atom(C,Class),
        concat(M," should have been defined in a subclass of ",
            E1),
        concat(E1,C,E),
        Self <- [error,E,[Self,Method]]
    )
]).

```

```

/* kind_of/1 */
ivar(objects,method,
[
  [kind_of,Self,Class],
  (
    ivar(Self,class,Class)
  )
]).
ivar(objects,method,
[
  [kind_of,Self,Class],
  (
    ivar(Self,class,C),
    C *- [sort_of,Class]
  )
]).

/* error/2 */
ivar(objects,method,
[
  [error,_,Message,Context],
  (
    stdio(Si,So),
    open(soap_err,9:10,8:60,w:red,"SOAP ERROR"),
    stdio(soap_err),
    write(Message),
    cursor(soap_err,3:0),
    write("Context: "),
    write(Context),
    cursor(soap_err,5:41),
    getkey(_),
    stdio(Si,So),
    break,
    close(soap_err),
    !,fail
  )
]).

/* self_destruct/0 */
ivar(objects,method,
[
  [self_destruct,Self],
  (
    /* undo dependance links */
    Self <- [broadcast,destroyed],
    forall(ivar(Self,dependant_on,D),
    Self <- [remove_dependant,D]), /* hook */
    /* remove Self as instance */
    ivar(Self,class,Class),
    Class <- [remove_instance,Self], /* hook */
    forall(retract(ivar(Self,_)),true)
  )
]).

```


/* add_dependant/1 */

```
ivar(objects,method,
[
    [add_dependant,Self,Dep],
    (
        Self +<- [dependant,Dep]
    )
]).
```

/* remove_dependant/1 */

```
ivar(objects,method,
[
    [remove_dependant,Self,Dep],
    (
        Self -<- [dependant,Dep]
    )
]).
```

/* any_atom/1

/* any_atom/1 */

```
ivar(objects,method, /* verifier */
[
    [any_atom,_,Term],
    (
        atom(Term)
    )
]).
```

/* any_positive_integer/1 */

```
ivar(objects,method, /* generator */
[
    [any_positive_integer,_,Term],
    (
        a_number(Term)
    )
]).
```

/* any_given_number/1 */

```
ivar(objects,method, /* verifier */
[
    [any_given_number,_,Term],
    (
        number(Term)
    )
]).
```

/* any_class/1 */

```
ivar(objects,method, /* generator */
[
    [any_class,_,Class],
    (
        classes *<- [has_instance,Class]
    )
]).
```

```

    ).

/* any_instance/1 */
ivar(objects,method,    /* generator */
    [
        [any_instance,_,Instance],
        (
            objects *← [has_instance,Instance]
        )
    ])

/* any_kind_of/1 */
ivar(objects,method,    /* generator */
    [
        [any_kind_of,_,I,Class],
        (
            Class *← [subsumes,Sort],
            ivar(Sort,instance,I)
        )
    ])

/* any_ivar_definition/1 */
ivar(objects,method,    /* verifier */
    [
        [any_ivar_definition,Self,ivar_def],
        (
            ivar_def = (Ivar_name::Def),
            Def =.. [Known_or_not,Card,Domain],
            (Known_or_not == known ; Known_or_not == can_be),
            (var(Card) ; integer(Card),Card > 0),
            (var(Domain) ; atom(Domain) ; list(Domain) ;
            tuple(Domain)),
            /* ivar's name can not be confused with any LOCAL
            method's name (it will shadow inherited methods of
            the same name) */
            not( ivar(Self,method,[Ivar_name,_,_],_) )
        )
    ])

/* any_method/1 */
ivar(objects,method,    /* verifier */
    [
        [any_method,Self,Clause],
        (
            Clause = [[Name,Var[Args],_],
            var(Var),
            list(Args),
            /* method name can not be confused with an (inherited)
            ivar name */
            not (Self *← [sort_of,Sort],
            ivar(Sort,ivar,Name::_))
        )
    ])

/* send/2 */

```

/* An object is asked to send a deterministic message */
ivar(objects,method,

```
[  
  [send,Self,Recipient,Message],  
    (  
      (list(Message),  
        /* in pre-processed format */  
        Recipient <- Message  
      ;  
      Message =.. Message1,  
      /* needs to be pre-processed */  
      Recipient <- Message1  
    ),!  
  )  
]).
```

/* CLASSES

with ivars: ivar,instance,superclass,subclass,method

**with methods: add_subclass/1, add_method/3, remove_method/1,
 add_ivar/2,remove_ivar/1,remove_instance/1,
 remove_subclass_link/1,default/2,
 cardinality/2,in_domain/2, self_destruct/0, print_string/1,
sort_of/1,subsumes/1, has_instance/1
*/**

**/* the ivar,instance,superclass,subclass and method instance variables are circularly inherited
since CLASSES is an instance of CLASS_CLASSES which is a subclass of CLASSES */
ivar(classes,ivar,ivar::known(_,any_ivar_definition)). /* strange loop*/
ivar(classes,ivar,instance::known(_,any_atom)).
ivar(classes,ivar,superclass::known(1,any_class)).
ivar(classes,ivar,subclass::known(_,any_atom)).
ivar(classes,ivar,method::known(_,any_method)).
/* */
ivar(classes,superclass,objects).
ivar(classes,subclass,class_objects).
ivar(classes,subclass,metaclasses).
ivar(classes,class,class_classes).**

/* methods */

**/* add_subclass/1 */
ivar(classes,method,
 [
 [add_subclass,Self,Name],
 (
 /* not an object already */
 not ivar(Name,_,_),
 /* give it a properly connected metaclass, thus making
 it an object */
 ivar(Self,class,Meta),
 Meta <- [new,Name],
 /* connect it */
 assert(ivar(Self,subclass,Name)),
 assert(ivar(Name,superclass,Self))
)
]).**

**/* remove_subclass_link/1 */
/* removes both subclass and associated metaclass connections */ ivar(classes,method,
 [
 [remove_subclass_link,Self,Name],
 (
 retract(ivar(Self,subclass,Name)),
 ivar(Self,class,Meta),
 ivar(Name,class,SubMeta),
 retract(ivar(Meta,subclass,SubMeta))
)
])**

```

    ).
/* add_method/1 */
ivar(classes,method,
[
    [add_method,Self,Method],
    (
        Self +<- [method,Method]
        ;
        Self <- [error,"Invalid method!",[Self,Method]],!
    )
]).

/* remove_method/1 */
ivar(classes,method,
[
    [remove_method,Self,Pred/N],
    (
        list_of_length(N,L),
        Head = [Pred,_|L],
        forall(Self <- [method,[Head,_]],true)
    )
]).
ivar(classes,method,
[
    [remove_method,Self,Pred],
    (
        Pred \= /(_,_),
        Head = [Pred,_|_],
        forall(Self <- [method,[Head,_]],true)
    )
]).

/* remove_instance/1 */
ivar(classes,method,
[
    [remove_instance,Self,Instance],
    (
        retract(ivar(Self,instance,Instance))
    )
]).

/* add_ivar/1 */
ivar(classes,method,
[
    [add_ivar,Self,Name::Def],
    (
        (
            ivar(Self,ivar,Name::_),!,
            Self <- [error,"Can't add ivar; it already exists!",
                [Self,Name,Def]]
        )
        ;
        (
            Self +<- [ivar,Name::Def]
        )
    )
]

```

```

;
Self <- [error,"Illegal ivar definition!",
        [Self,Name,Def]]
),!
),!
)
)).

/* default/2 */
ivar(classes,method,
[
  [default,Self,Ivar_name,Default],
  (
    ivar(Self,ivar,Ivar_name::Def), /* its local */
    Def =.. [can_be,_,Domain], /* if can be defaulted */
    (var(Domain)
    ;
    (list(Domain),member(Default,Domain)
    ;
    Domain =.. [Pred|Args],
    Generator = [Pred,Default|Args],
    Self *-< Generator
    )
    )
    )
  )
)).
ivar(classes,method,
[
  [default,Self,Ivar_name,Default],
  (
    /* it must be inherited */
    not (ivar(Self,ivar,Ivar_name::_)),
    ivar(Self,superclass,Super),!,
    Super *-< [default,Ivar_name,Default]
    )
  )
)).

/* in_domain/2 */
ivar(classes,method,
[
  [in_domain,Self,Ivar_name,Value],
  (
    ivar(Self,ivar,Ivar_name::Def), /* it's local */
    Def =.. [_,_,Domain],
    (var(Domain)
    ;
    (list(Domain),member(Value,Domain)
    ;
    Domain =.. [Pred|Args],
    Verifier = [Pred,Value|Args],
    Self <- Verifier
    )
    )
    )
  )
)).

```

```

    ]).
ivar(classes,method,
[
    [in_domain,Self,ivar_name,Value],
    (
        /* it must be inherited */
        not (ivar(Self,ivar,ivar_name:::)),
        ivar(Self,superclass,Super),!,
        Super <- [in_domain,ivar_name,Value]
    )
]).

/* self_destruct/0 */
ivar(classes,method,
[
    [self_destruct,Self],
    (
        /* destroys all instances */
        forall(ivar(Self,instance,I), I <- [self_destruct]),
        /* destroys itself as a class */
        ivar(Self,superclass,Super),
        ivar(Self,class,Meta),
        ivar(Super,class,SuperMeta),
        /* reconnects subclasses and their metaclasses */
        forall(ivar(Self,subclass,Sub),
            (
                Sub <- [new_superclass,Super],
                ivar(Sub,class,SubMeta),
                SubMeta <- [new_superclass,SuperMeta] )),
        /* destroys itself as object */
        Super <- [remove_subclass_link,Self],
        Self <- [self_destruct] @ classes,
        Meta <- [self_destruct] /* metaclass destruction */
    )
]).

/* new_superclass/1 */
ivar(classes,method,
[
    [new_superclass,Self,Super],
    (
        /* disconnects and reconnects itself */
        retract(ivar(Self,superclass,OldSuper)),
        retract(ivar(OldSuper,subclass,Self)),
        assert(ivar(Self,superclass,Super)),
        assert(ivar(Super,subclass,Self))
    )
]).

/* print_string/1 */
ivar(classes,method,
[
    [print_string,Self,Ps],
    (

```

```

                                concat("a class named ",Self,Ps)
                                )
        }).

/* cardinality/2 */
ivar(classes,method,
    [
        [cardinality,Self,ivar_name,Card],
        (
            ivar(Self,ivar,ivar_name::Def),
            Def =.. [_Card,_]
        )
    ]).
ivar(classes,method,
    [
        [cardinality,Self,ivar_name,Card],
        (
            not ivar(Self,ivar,ivar_name::Def),
            ivar(Self,superclass,Super),!,
            Super <- [cardinality,ivar_name,Card]
        )
    ]).

/* sort_of/1 */
ivar(classes,method,
    [
        [sort_of,Self,Self],
        (
            true
        )
    ]).
ivar(classes,method,
    [
        [sort_of,Self,Sort],
        (
            ivar(Self,superclass,Super),!,
            Super *-< [sort_of,Sort]
        )
    ]).

/* subsumes/1 */
ivar(classes,method,
    [
        [subsumes,Self,Self],
        (
            true
        )
    ]).
ivar(classes,method,
    [
        [subsumes,Self,Subs],
        (
            ivar(Self,subclass,Sub),
            Sub *-< [subsumes,Subs]
        )
    ]).

```



```

    )
  }).

/* has_instance/1 */
ivar(classes,method,
  [
    [has_instance,Self,Inst],
    (
      Self * <- [subsumes,Class],
      ivar(Class,instance,Inst)
    )
  ]).

```

/* METACLASSES

with ivars: none yet

***with methods: new/1, self_destruct/0
*/***

***ivar(metaclasses,superclass,classes).
ivar(metaclasses,class,class_metaclasses).
ivar(metaclasses,instance,class_metaclasses).
ivar(metaclasses,instance,class_objects).
ivar(metaclasses,instance,class_classes).***

/* methods */

/* new/1 */

***/* Given a class, creates the corresponding metaclass and connects it as
its subclass */***

***ivar(metaclasses,method,
[
[new,Self,Name],
(
concat('class_',Name,Meta1),
string_atom(Meta1,Meta),
assert(ivar(Self,subclass,Meta)),
assert(ivar(Meta,superclass,Self)),
assert(ivar(Meta,instance,Name)),
assert(ivar(Name,class,Meta)),
assert(ivar(metaclasses,instance,Meta)),
assert(ivar(Meta,class,metaclasses))
)
])).***

/* self_destruct/0 */

/* simple self_destruction */

***ivar(metaclasses,method,
[
[self_destruct,Self],
(
(retract(ivar(Self,_,_))),fail
;
true),
retract(ivar(metaclasses,instance,Self))
)
])).***

/* CLASS_CLASSES :

with ivars: <none>

with methods: <none>

***/**

ivar(class_classes,superclass,class_objects).

ivar(class_classes,instance,classes).

ivar(class_classes,class,metaclasses).

ivar(class_classes,subclass,class_metaclasses).

/* CLASS_OBJECTS

with ivars: <none>

**with methods: new/1, all_instances/1,remove_all_instances/0
*/**

**ivar(class_objects,class,metaclasses).
ivar(class_objects,superclass,classes).
ivar(class_objects,subclass,class_classes).
ivar(class_objects,instance,objects).**

/* methods */

/* new/1 */

**ivar(class_objects,method,
[
 [new,Self,Name],
 (
 Self <- [new_name,Name],
 /* put in new object (avoid message passing) */
 asserta(ivar(Name,class,Self)),
 asserta(ivar(Self,instance,Name))
)
]).**

/* new_name/1 */

**ivar(class_objects,method,
[
 [new_name,Self,Name],
 (
 var(Name),
 Self * <- [gensym,Name],
 not ivar(Name,_)
)
]).**

**ivar(class_objects,method,
[
 [new_name,Self,Name],
 (
 nonvar(Name),
 not ivar(Name,_)
)
]).**

/* gensym/1 */

**ivar(class_objects,method,
[
 [gensym,Self,Name],
 (
 a_number(N),
 string_atom(S1,Self),**

```

concat("a_kind_of_",S1,S2),
string_integer(S3,N),
concat(S2,S3,S4),
string_atom(S4,Name)
)

)).

/* all_instances/1 */
ivar(class_objects,method,
[
    [all_instances,Self,List],
    (
        bagof(I,Self *:- [has_instance,I],List)
    )
]).

/* remove_all_instances/0 */
ivar(class_objects,method,
[
    [remove_all_instances,Self],
    (
        forall(Self *:- [has_instance,I],
            I <- [self_destruct])
    )
]).

```

/* CLASS_METACLASSES

with ivars: none yet
with methods: none yet
***/**

ivar(class_metaclasses,superclass,class_classes).
ivar(class_metaclasses,class,metaclasses).
ivar(class_metaclasses,instance,metaclasses). /* circularity */

*<- /2	94	has_instance/1 (classes)	116
++<- /2	95	image (saving, loading)	92
+<- /2	95	inc_state/1	97
-+<- /2	95	instance_variable/1 (objects)	105
-<- /2	95	in_domain/2 (classes)	113
<- /2	94	ivar_of/2	96
=<- /2	94	kind_of/1 (objects)	107
==<- /2	94	list_of_length/2	97
@/2 (method definition locator)	93	METACLASSES	117
add_dependant/1 (objects)	108	new/1 (class_objects)	119
add_instance_variables/2	98	new/1 (metaclasses)	117
add_ivar/1 (classes)	112	new_name/1 (class_objects)	119
add_method/1 (classes)	112	new_superclass/1 (classes)	114
add_methods/4	98	OBJECTS	103
add_subclass/1 (classes)	111	print_string/1 (classes)	114
all_instances/1 (class_objects)	120	print_string/1 (objects)	106
any_atom/1 (objects)	108	put_value/2 (objects)	103
any_class/1 (objects)	108	remove_all_instances/0 (class_objects)	120
any_given_number/1 (objects)	108	remove_dependant/1 (objects)	108
any_instance/1 (objects)	109	remove_instance/1 (classes)	112
any_ivar_definition/1 (objects)	109	remove_method/1 (classes)	112
any_kind_of/1 (objects)	109	remove_subclass_link/1 (classes)	111
any_method/1 (objects)	109	replace/4	96
any_positive_integer/1 (objects)	108	responds_to/1 (objects)	106
a_number/1	96	save_all_objects/0	101
begins/2	97	save_soap/0	101
break_clause/3	97	self_destruct/0 (classes)	114
broadcast/1 (objects)	105	self_destruct/0 (metaclasses)	117
cardinality/2 (classes)	115	self_destruct/0 (objects)	107
cardinality_violation/1 (objects)	104	send/2 (objects)	110
changed/0 (objects)	105	should_not_implement/2 (objects)	106
CLASSES	111	signal/2 (objects)	105
CLASS_CLASSES	118	soap_arrow/1	100
CLASS_METACLASSES	121	soap_class/3	98
CLASS_OBJECTS	119	soap_class/4	98
clear_all/0	92	soap_method/3	98
count_clause1/2	97	soap_translate/4	99
count_clauses/2	97	sort_of/1 (classes)	115
debug/1	100	subsumes/1 (classes)	115
debug_soap/1	100	translate_conj/4	99
deep_replace/4	96	translate_all/4	100
default/2 (classes)	113	translate_message/3	99
delete/3	97	update/0 (objects)	105
do/1	100	update/1	96
do_do/2	100	view_debug_file/1	101
ends/2	97	write_message/1	101
error/2 (objects)	107	write_stack_usage/0	101
failure/0	101	yourself/1 (objects)	103
fire_method/2	96		
first/3	97		
gensym/1 (class_objects)	119		
get_value/2 (objects)	104		
get_value_no_default/2 (objects)	105		