NClosEmacs: An Emacs-based Expert System Fran

jmc

Overview

This document describes NClosEmacs, a tentative implementation of the NClose rules engine in ELisp for embedding expert system functionality in the Emacs environment.

In addition, this implementation is experimental in that it investigates the use of *modern* programming approaches in this new implementation, most of which were only budding when NClose itself was designed in the early eighties.

As such, it is still with a certain pride, after all these years, that I qualify the following work as research in progress.

1 Introduction and historical background

NClose is an original evaluation algorithm designed for research on rule-based systems, with which specific domain applications in Medicine were explored in the mid-eighties, at the Robotics Institute, Carnegie-Mellon University.

At that time NClose emerged as a first result in an effort to pursue the development of computer programs capable of displaying (hopefully) intelligent problem-solving behaviour. This approach to Artificial Intelligence was unsurprisingly inspired from the prevalent school of thought at CMU led by Allen Newell and Herbert Simon. Work by Newell, "The Knowledge Level" in particular, but research at MIT driven by Pete Szolovits, "Artificial Intelligence in Medicine", for instance, and at Stanford with authors such as Buchanan, Feigenbaum, Shortliffe and Fagan also proved inspirational, specifically in relation to domain applications.

A second angle which drove these early research effort was the realisation that “rules” or “productions” could be envisioned as both representing *computation* elements, within a computer science perspective, and *knowledge* units, in the larger cognitive sciences framework which was budding from AI research at the time. This duality of views in itself provided an exploratory bridge between computing processes and cognitive behaviour which, with the upcoming availability of PCs and graphical user interfaces, readily lent itself to an experimental mode of research.

From the late eighties to the late nineties, this experimental mode would morph into more implementation-oriented work, culminating in the very successful releases of a series of commercial tools and environments at the front edge of the then nascent major trends in an emerging software industry: graphical user interfaces and object-oriented programming languages, which are both now firmly established as our common base infrastructure.

Over time several implementations of the base NClose algorithm were developed in various programming languages including LISP (MacLisp, LeLisp, and ZetaLisp chiefly), and later Pascal, C and C++. NClose evaluation constituted the core of the rule engine found in the commercial products Nexpert, and Nexpert Object, released by Neuron Data from 1985 onward.

2 NClose in Emacs

This package contains a preliminary implementation of NClose in ELisp, the dialect of the Lisp programming language used in both GNU Emacs and XEmacs (collectively called Emacs in this document). Users of Emacs commonly write ELisp code to extend and to customize Emacs as they require.

This document does not dwell on the NClose algorithm itself, a formal description of which can be found in Rappaport and Chauvet, 1984, and rather focuses on several facets of its implementation in ELisp, with emphasis on the experimental approach to this revisited development.

This implementation departs from the somewhat older implementations in early pre-Common Lisp dialects (namely MacLisp and ZetaLisp) which was leveraged in our AI research work. In particular, ELisp offers "advising functions" which are useful for Aspect Oriented Programming (AOP). AOP, which originated with Gregor Kiczales at Xerox PARC, helps programmers separate so-called cross-cutting concerns, i.e. functionality which has an impact on all the functions of a program, from the program code itself. (Logging is the traditional example of a cross-cutting concern.) The AOP flavor provided by advising functions in Emacs prompted revisiting the NClose algorithm with an eye towards proper identification of cross-cutting concerns.

Another revision of the original design ideas behind NClose was deemed necessary as Emacs is, for all purposes of this implementation, the unique communication channel with the end-user. As one of the distinguishing features of NClose, and the ensuing Nexpert implementations and commercial products, is precisely its handling of user interactions, the embedding into Emacs both restricted and enhanced what we used to implement in windowing environments and graphical user interfaces. Living in Emacs, text naturally becomes the primary user interface through which commands are issued and knowledge bases loaded and run. Special buffers play roles otherwise devolved to windows or menus in previous implementations. New usages may be envisioned where knowledge-driven problem solving seamlessly blends into text documents produced by the user.

The following sections provide an overview of the design options selected in the current NClosEmacs implementation.

2.1 ELisp Implementation

The main implementation file is '`nclose.el`'. It loads the various ELisp files required, installs a new major mode for NClose rule bases and inits the rules engine.

Each large functional chunk is implemented in its own ELisp file named somewhat after its main role in the NClose algorithm. More detailed explanations may be found in See Chapter 5 [Implementation], page 21.

NClosEmacs relies on the ELisp evaluator for its rules evaluation engine. Each piece of data in the left- and right-hand sides of rules, usually called *signs*, *variables* or simply *data* in the rest of the document, and each *hypothesis* is actually implemented as an ELisp symbol. This means that the value of a sign is obtained by typing its name at the top level Lisp listener. For instance: '`ACETATE-CYPROTERONE`' and '`(eval 'ACETATE-CYPROTERONE)`' both return the value of the sign ACETATE-CYPROTERONE. Unknown signs are simply unbound ELisp symbols.

Another nice consequence of the reuse of the evaluator is that conditions patterns in left-hand sides may be expressed directly in Lisp. Similarly actions in the righ-hand sides may use Lisp functions directly. Pattern evaluation uses the error signalling mechanism of ELisp to trigger backward chaining if the unbound sign encountered is in fact an hypothesis, or a question to the user if simply a sign. In ‘unify.el’ the basic LHS evaluation is as follows:

```
(defun nclose-get-unification (pattern)
  "Interactive unification of pattern"
  (condition-case error
      (eval pattern)
    (void-variable
     (sign-writer (cadr error) (sign-reader (cadr error)))
     ;; Recurse
     (nclose-get-unification pattern)
    )
    )
)
```

The idea is that when the (Lisp) evaluation of the pattern fails on an unbound symbol, the error catching mechanism considers it as an unknown sign or hypothesis and accordingly assigns a value read possibly interactively, and retries. This process recursively evaluates all required signs in the LHS to produce the final unification result, here a true or false value.

The ‘sign-writer’ and ‘sign-reader’ high level getter/setter functions for signs are separately coded as they may be advised in order to implement side-effects.

Similarly a rule RHS is passed to the Lisp evaluator for execution with the sequence of Lisp forms implicitly and-ed, as in:

```
(defun and-eval-rhs (rhs)
  "Recursively execute the sequential execution of RHS forms"
  (cond
   ((null rhs) t)
   (t (and (eval (car rhs)) (and-eval-rhs (cdr rhs))))
   )
)
```

This works provided all RHS operators are pre-defined to call the advised functions if necessary. This is in particular the case for the ubiquitous ‘@SET’ operator which expects exactly a sign as its first argument and a value as the second argument. In the file ‘rhs.el’, this operator, for instance, is implemented as a Lisp macro:

```
(defmacro @set (var val)
  "External RHS assignment operator"
  (progn (sign-writer var (nclose-get-unification val)) t)
)
```

This example implementation of a RHS operator shows the important features to keep in mind. Execution indeed returns a boolean value; remember the implicit and in the sequence of RHS forms execution. Returning '`nil`' would stop further execution of the RHS. (This crude control mechanism could be useful to an advising function for instance.)

The RHS value assignment itself is a two-step process: the value is computed, which may entail asking the user for input, hence the call to the generic '`nclose-get-unification`' evaluation/unification function also used in LHS patterns evaluation; and then it is assigned to the said sign through the standard setter, which as previously mentioned may itself be advised.

Other domain-specific operators may be developed by following the same simple design method. An interesting possibility is, of course, to call Emacs' text and buffer functions from the RHS of NClose rules, opening up a whole scope of applications with text documents co-produced, in form or content or both, by the user and a rule-based expert system.

2.2 Aspect Oriented Programming

The experimental part of this new implementation of NClose relies on using Aspect Oriented Programming ideas, and more specifically a recent evolution known as Context Oriented Programming, where different cross-cutting concerns are implemented in *layers* separately activated as needed.

Although not all features of NClose were yet reviewed for this version of the implementation, at least two were immediately identified as appropriate candidates for an AOP approach:

- Gating: the core NClose forward chaining process
- Logging: keeping a trace of the expert system session execution

The "gating" process in NClose is a form of forward chaining, going from known signs or data to hypotheses. In contrast to RETE-like implementations where individual conditions in separate rules are compiled into a automate network, with known data acting as tokens moving from one vertex to another as they match the condition node, the somewhat simpler implementation in NClose uses a "gate", which, when a sign becomes known, decides which hypotheses should be posted for later evaluation.

In order to implement gating, a gating-layer is created as a series of advising functions in ELisp. Given that the core backward chaining is implemented through an agenda, a last-in first-out stack of hypotheses posted for evaluation in bacward chaining mode, the gating-layer addresses concerns:

- initialization time, for the setup of the data structures responsible for handling gating;
- rule compilation, to handle dependencies between hypotheses and signs discovered when parsing LHSes;
- execution time, each time a value is assigned to a piece of data, i.e. when it becomes known.

The layer is hence implemented as three advising functions, gathered with other layers in the '`advice.el`' file. The following advising function, for instance:

```
(defadvice sign-compile (after gating-layer (sign hypo))
  "Builds the forward association list, hypo is nil when RHS are compiled"
  (if hypo (plist-push sign hypo))
)
```

advises the '`sign-compile`' function, called at rule compilation time, and adds the rule's hypothesis to the parsed sign forward association list. This forward association list is later used, at runtime, to push hypotheses on the agenda when signs become known.

In order to implement a minimal logging facility, another layer is implemented in '`advice.el`'. The functions in this layer simply advise all of the interactive command functions, keeping track of all interactions with the user, as well as the evaluation functions, keeping trace of the expert system's deductions as the session progresses. These traces are stored in a special Emacs buffer named '`*nclose-log*`' and addressable through the usual set of Emacs buffer commands.

Advising functions are enabled and activated by default in the same file:

```
;; Enable/Disable layers as required
(ad-enable-regexp "gating-layer")
(ad-enable-regexp "log-layer")

;; Activate advising functions
(ad-activate-regexp "gating-layer")
(ad-activate-regexp "log-layer")
```

other variations are of course possible, since Emacs' advising functions may be enabled and disabled using regexp patterns rather than names.

2.3 Front End

The front-end of this NClose implementation is unsurprisingly text-oriented and well-integrated with Emacs. The purpose of the default front-end is to assist in the authoring and running of rule bases from within Emacs.

The NClose rules engine is manually loaded by issuing the standard Emacs command '`M-x load-file`' on the master NClose '`nclose.el`' file (with the .elc extension if it has been compiled at installation). This may also be placed in the '`.emacs`' initialization file.

The rule base itself is a text document edited in a standard Emacs buffer. See Chapter 3 [Authoring Knowledge Bases], page 8, for the actual syntax of rules declarations. The user "loads" a rule base in memory simply by evaluating the buffer containing its text, the '`M-x eval-current-buffer`' in Emacs.

NClose provides a number of commands, implemented as interactive ELisp functions invoked through the usual '`M-x`' prefix to setup and run an expert session. In addition, traces and execution logs may be provided in NClose-specific buffers such as '`*nclose-log*`'.

The '`nclose.el`' file also provides a minimal major mode for text-based rule bases. At this stage, the major mode is only concerned with highlighting the inference process as it progresses through the rule base. Hypotheses under evaluation are highlighted as they are pushed and popped from the agenda; their final status, true or false, is color-coded. NClose keywords are also font-locked for easier readability. (The NClose major mode is still pretty much experimental and additional features may be added as the implementation is revised.)

3 Authoring Knowledge Bases

Rule bases for NClose are simple text documents which can be edited in Emacs. These rule base text documents follow the Lisp syntax conventions, and more specifically the ELisp conventions. Comments lines begin with the '`;`' character. The body of the rule base is constituted of well-balanced Lisp forms, one for each rule in the knowledge base.

3.1 Writing rules

NClose provides a high-level Lisp macro to declare a rule.

```
(add-to-kb
  (@LHS= *pattern-lisp-form*)
  (@hypo *hypothesis*)
  [(@RHS= [*RHS-lisp-form*]+) *string-documentation*]
)
```

The macro parses four arguments, the last two being optional:

- *pattern-lisp-form* is a lisp form expressing a boolean computation with an ultimate true/false value. It usually is an AND/OR tree of several elementary conditions involving any of the standard Lisp functions, namely: '`(string= < > = /= >= <= + - * / and or not null yes no Yes No)`'. Any other symbol in the pattern is considered as sign and compiled as such by the rule compiler.
- *hypothesis* is the hypothesis for the rule.
- Optionally, *RHS-lisp-form* is one of possibly several right-hand side actions. Currently the only NClose specific RHS action is the '`(@SET *sign* *expression*)`' operator used to assign a value (*expression*) to a sign. Other Lisp functions in a RHS action are passed without modification to the Lisp evaluator.
- *string-documentation* is an optional textual description attached to the rule.

For instance the following are examples or well-formed, if meaningful, rules:

```
(add-to-kb (@LHS= (and (> a 2) (string= foo "hello"))) (@hypo H3)
(@RHS= (@set frob (* 2 a)) (@set boz "test")) "Another
comment")

(add-to-kb (@if (> boz 0)) (@hypo H3))

(add-to-kb (@if (and (> a c) (< (* d 5) 10))) (@hypo H2)
nil "Commenting this H2 rule")
```

3.2 Designing classes and objects

Still an area of research and experimental implementation work.

The introduction of a simple yet rich object system in Nexpert was the major transition between the original version and the later Nexpert Object. This transition happened as early as 1986-87, at a time when object-oriented programming languages were beginning to vie for developers' attention and frame-based and object systems were still an active area of AI research.

While object-oriented languages became mainstream, object systems faded from the main AI research efforts only to find renewed attention in so-called *middleware*, with COM/DCOM from Microsoft and the Object Management Group's Corba/IIOP specifications being the major industry beacons during the early nineties. Although these were somewhat superseded later on in the decade by the emergence of Web protocols-based middleware using XML (SOAP/WSDL and the plethora of WS- specifications) they played role model in the development of the Web Services stack.

Today, with the Semantic Web initiative at the W3C—albeit making slow progress in industry adoption—object systems are again under active study and development. The crux of the on-going Semantic Web matter lies in the definition and use-cases of *ontologies*. Proposals differ on the expressivity of ontology specifications, on their actual representation, and on their articulation with deductive systems of various kinds—mostly based on first-order logic. Two important ontology systems in broad use today are the W3C-promoted RDF/OWL and Topic Maps. Another unrelated initiative, dubbed *microformats*, is more interested in in-page data formatting for automated collection and analysis.

3.2.1 The NClosEmacs articulation between rule and object systems

The important innovation in the transition from Nexpert to Nexpert Object was in the elegant articulation between the rule and the object systems. On the one hand the object system supported *classes*, defined as collections of *properties*, and inheritance along two relations: *subClassOf* and *instanceOf*, respectively from class to class(es) and from class to object(s). A third relation, from object to object(s), defined as *partOf*, allowed two hierarchies (class to subclasses with objects as leaves, and object to subobjects) to mingle in Nexpert ontologies.

The system was designed as quite dynamic in nature and weakly reinforced hierarchical relations, in contrast to object-oriented programming languages—and more in par with the then AI-styled, and now Semantic Web research. Objects could sport additional properties to the set of inherited ones; exceptions in default inheritance mechanisms could be provided; objects could be moved from class to class; objects could be created by rules' RHS actions, and so forth.

The articulation between rules and objects basically enabled LHS of rules to query the object system for a set of subsets of objects matching simple patterns and RHS to act upon this set of subsets. The syntax for expressing patterns was very compact and represented classes as set of their instances, and objects as sets of their subobjects with LHS patterns being either set operations such as membership or intersections, or scalar typed tests on properties.

In the NClosEmacs proposed implementation we are trying to abstract out simple design principles from the original ideas in Nexpert Object to layer an object system over the NClose rule engine.

- The "static" part of the object system, the base ontology, should minimally provide classes and objects with two hierarchical relations (namely class/subclass and object/subobject, usually specialization and mereological). Inheritance mechanisms are overlayed on the object system ontology.

- The "dynamic" part of the object system, is the definition of the query language (patterns) used in rules LHSes and the object-related operations in the rules RHSes.

Note that, in addition, the unification step should now collect set of subsets of objects, or list of *scopes*, to be passed from the LHS to the RHS.

In the vein of contemporary systems we are interested in using NClosEmacs knowledge-based applications on modern ontology representations. So we plan again to separate concerns and provide for different ontology representations together with an abstract API for user-defined ontology representations—which may be required for domain-specific applications, e.g. health/medicine applications.

In NCLosEmacs the interchangeable ontology modules are called *nclos* for NClosEmacs Ontology/Object System.

NClosEmacs comes with three ontology modules:

- A default ontology with no inheritance which relies heavily on the underlying Lisp evaluator and specifically on atom property lists, See Section 3.2.2 [The Default Object System in NClosEmacs], page 10
- An ontology representation based on OWL-Lite where classes and objects are described using the OWL-Lite conventions, See Section 3.2.3 [The OWL-Lite Object System in NClosEmacs], page 15
- An ontology representation based on Topic Maps where classes and objects of NClosEmacs are represented as such.

Both OWL-Lite and XTM (Topic Maps) ontology representations offer a much more expressive environment for ontologies than the original Nexpert Object object system. The current implementation is only a basis for further research work inasmuch as the minimal feature set for NClosEmacs was imported from OWL-Lite and Topic Maps. Augmenting the pattern/action language to leverage their additional expressive power constitute an open area for further research. (The implementation could use Emacs advising functions, for instance.)

3.2.2 The Default Object System in NClosEmacs

The default object system in NClosEmacs uses the underlying ELisp variables and evaluator to represent, store and access objects. Individual objects are ELisp variables, and their individual properties and values are stored in the symbol's property list itself. Classes are not distinguished from composite objects (superobjects) in this default nclos; there is no inheritance or derivation mechanism. Classes are using lists as simple set representations.

The default nclos is installed by invoking:

```
(nclose-use-nclos nil)
```

at the beginning of the knowledge base file.

Note that by default the default nclos is installed the first time NClosEmacs is loaded. After the default nclos installation, classes and objects are simply defined by the add-to-kb-sets statement in the knowledge base:

```
(add-to-kb-sets 'Ca  '(O1 O2 O3))
(add-to-kb-sets 'Cb  '(O4 O5))
```

These two statements define two classes, “Ca” and “Cb” respectively. The first class has three instances, namely objects “O1”, “O2” and “O3”. The class Cb has two instances: objects “O4” and “O5”.

In the default nclos objects' properties do not require definitions as the properties and the objects they attach to are actually derived from the rule parsing phase. As a consequence the simple ontology model supported by the default nclos is a collection of sets, or classes, of individual objects, with no inheritance but dynamical property lists for each object.

Writing Rules with Patterns in the default nclos

With each nclos systems comes a set of specific LHS query operators and a related set of RHS commands to be used in rules. In addition to matching LHS conditions, the inference engine is also required to *unify* patterns and subsets of objects in the ontology. A pattern in a rule LHS can be seen as a query against to the ontology which returns a, possibly empty, set of objects. Hence the unification of a LHS not only returns *true* or *nil* to indicate whether all of its conditions are matched or not, but, in addition, a *result set* accumulating all the subsets of objects that matched its patterns.

When the LHS is matched, this result set is passed to the rule RHS where object-specific commands can operate upon these sets and their individual object members.

In the default nclos there are basically two major patterns: *all* and *some*, which query an individual class for all its instances matching a given condition. The general form of a pattern is:

```
(all-in 'CLASS-NAME INTEGER 'AND-OR-ELISP-FORM)
```

and

```
(some-in 'CLASS-NAME INTEGER 'AND-OR-ELISP-FORM)
```

The first pattern (*universal*) is true when all instances of the said class match the lisp form passed as the last argument. Its result set is simply the list of instances of the said class.

The second pattern (*existence*) is true when some, i.e. at least one, instances of the said class match the lisp form passed as the last argument. Its result set is the subset of this class instances that actually matched the test lisp form.

Note the integer second argument passed in both patterns. This is the index of the subset in the rule global result set, on which the query should be applied. Several patterns in the same rule LHS may refer to the same index; they then act as successive filters on the subset resulting on previous pattern-matching queries on the same index.

In the third argument AND-OR-ELISP-FORM, each free ELisp variable is considered a *property*, defined on all instances of the class passed as the second argument. These “dynamic” properties are created and added to objects’ representations on an as-needed basis, by adding them to the ELisp variable symbol property list on the fly.

Let’s look at a few examples.

```
(add-to-kb-sets 'Ca  '(O1 O2 O3))
(add-to-kb-sets 'Cb  '(O4 O5))

(add-to-kb
 (@if (and (> x 0)
   (and (all-in 'Ca 1 '(or (> p1 0) (> p2 0)))
(some-in 'Ca 1 '(> p3 5)))))
 (@hypo H)
)
```

This knowledge base defines two classes, Ca and Cb, as before and adds one rule with a LHS containing two patterns, the first one being an universal pattern on class Ca and result set number 1, the second one being an existence pattern, also on class Ca and result set number 1.

The ELisp free variables *p1* and *p2* appear in the first pattern, and *p3* in the next one. Hence these three properties may be attached to instances of the the class Ca, namely objects *O1*, *O2*, and *O3*. In contrast the variable *x* is not bound in any pattern and hence not a property of an individual object, simply a scalar—here numeric—variable.

This example rule fires when:

- the variable *x* is positive, and
- both (i) all instances of class Ca have either a positive *p2* or a positive *p1*, and (ii) some of these same instances have a *p3* greater than 5.

The unification succeeds if these conditions are met and returns a result set containing only one subset. This unique subset is made of all instances of Ca with a *p3* greater than 5, and, by construction, either *p1* or *p2* positive.

For instance, let’s suppose the ontology looks like the following table:

Ca	**O1**	**O2**	**O3**
p1	12	-5	2
p2	5	3	-1
p3	6	0	12

Provided *x* is positive, the first pattern is matched: all instances of class Ca have either *p1* or *p2* positive. Only objects *O1* and *O3*, however, match the second pattern with *p3* greater than 5. Hence the LHS is true and the result set is the set:

```
((O1 O3))
```

for further use in the RHS, if any.

Now consider a slightly differently phrased rule:

```
(add-to-kb
 (@if (and (> x 0)
   (and (all-in 'Ca 1 '(or (> p1 0) (> p2 0)))
(some-in 'Ca 2 '(> p3 5)))))
 (@hypo H)
)
```

Because the second pattern now mentions a different index than the first one, the LHS is true of false under the same situation than the previous one, but its result set now contains two subsets rather than one. The first one is the list of (all) instances of class Ca with either a positive *p1* or a positive *p2*; the second one is the list of instances with a *p3* greater than 5:

```
((O1 O2 O3) (O1 O3))
```

These two subsets are indeed different as the following rule, slightly edited again, shows:

```
(add-to-kb
 (@if (and (> x 0)
   (and (some-in 'Ca 1 '(or (> p1 0) (> p2 0)))
(some-in 'Ca 1 '(> p3 5)))))
 (@hypo H)
)
```

We are back with a single result subset but now two existence patterns on the same subset. The LHS is now true only if among the instance of Ca with either a positive *p1* or a positive *p2*, some also have a *p3* greater than 5. The two succesive patterns act as filters, selecting a first subset of Ca's instances and refining this subset with the next selection. When the LHS is true the result set returns those instances that passed both filters.

Suppose the ontology looks like this new table:

Ca	**O1**	**O2**	**O3**
p1	12	-5	-2
p2	5	3	-1
p3	6	0	12

the result set of applying the later rule is:

```
((O1))
```

while the former rule would fail, the LHS would be set to false and the result set empty. (*Exercize: Why?*)

If the rule had been written with a different index for the second pattern, however:

```
(add-to-kb
 (@if (and (> x 0)
   (and (some-in 'Ca 1 '(or (> p1 0) (> p2 0)))
(some-in 'Ca 2 '(> p3 5)))))
 (@hypo H)
)
```

the result set for the later ontology table would contain two subsets:

```
((O1 O2) (O1 O3))
```

(*Same exercize: Why?*)

As a rule LHS can alternate as many universal and existence queries over the same or different result subsets, a rule may express fairly complex queries against the simple default nclos ontology.

Writing Rules with RHSes in the Default Nclos

As previously mentioned the unification result set is passed, when the rule is fired, to the RHS which basically executes its sequence of commands.

The syntax of the '`@set`' RHS command has been extended to handle the results of the LHS unifications. The familiar syntax is, as a reminder:

```
(@set VARIABLE SCALAR-VALUE)
(@set VARIABLE LISP-FORM)
```

which respectively assign a value or a computed value to the said variable. (This assignment may in addition trigger the forward chaining mechanism.)

Two new forms of the set command are available with the default nclos:

```
(@set (prop-in OBJECT-NAME PROPERTY-NAME) [ SCALAR-VALUE | LISP-FORM ])
(@set (member-in CLASS-NAME INTEGER PROPERTY-NAME)
                 [ SCALAR-VALUE | LISP-FORM ])
```

The first new form is used to assign a (computed) value to the designated property of the said object. For instance:

```
(@set (prop-in 'O1 'p3) 5)
```

sets the value of the *p3* property of object *O1* to the integer 5.

The second form is used to assign a (computed) value to the designated property of all objects in the subset, characterized by the class and index passed, in the LHS unification result set.

A rule with a RHS:

```
(add-to-kb
 (@if (and (> x 0)
   (and (some-in 'Ca 1 '(or (> p1 0) (> p2 0)))
(some-in 'Ca 1 '(> p3 5)))))
 (@hypo H)
 (@then
(@set y (+ x 1))
(@set (prop-in 'O3 'p4) (* 2 x))
(@set (member-in 'Ca 1 'p4) y)
 )
)
```

working against the simple aforementioned ontology:

Ca	**O1**	**O2**	**O3**
p1	12	-5	-2
p2	5	3	-1
p3	6	0	12

would evaluate its LHS to true, fire, execute the RHS side-effects resulting in a final ontology modified as follows:

Ca	**O1**	**O2**	**O3**
p1	12	-5	-2
p2	5	3	-1
p3	6	0	12
p4	x+1		2*x

(*Final exercize: Why?*) Note that *O2* has no assigned *p4* property after the rule fires. This is because *O2* is filtered out by all patterns in the LHS and not explictly mentioned in the RHS: it never "meets" the *p4* property.

Remark: In the current implementation the new set assignment commands do not trigger any form of forward chaining. This is an interesting avenue of research as there are several levels of granularity available for doing so: object level and class level namely.

3.2.3 The OWL-Lite Object System in NClosEmacs

The OWL-Lite nclos uses definitions and conventions from the OWL Lite specifications for representing ontologies. In this nclos, complete ontologies contain classes, objects, properties and slots. In contrast to the default nclos, the OWL-Lite nclos supports inheritance from class to subclasses.

Classes in the OWL-Lite nclos

A class defines a group of individuals that belong together because they share some properties. For example, Deborah and Frank are both members of the class Person. Classes can be organized in a specialization hierarchy using subClassOf. There is a built-in most general class named Thing that is the class of all individuals and is a superclass of all OWL classes. There is also a built-in most specific class named Nothing that is the class that has no instances and a subclass of all OWL classes.

Within an ontology, a class is defined by the following statement:

```
(owl-class! 'Cb 'Ca)
```

which defines the class '`Cb`' as a subclass of class '`Ca`'. By convention the rule of all classes in the OWL-Lite nclos is called '`'Thing`' and must be stated in every ontology as:

```
(owl-class! 'Thing nil) ;; Required
```

The graph of all classes is thus a tree rooted in "Thing".

Objects in the OWL-Lite nclos

In addition to classes, we want to be able to describe their object members. We normally think of these as individuals in our universe of things. An individual is (minimally) introduced by declaring it to be a member of a class.

The declaration statement for an individual in the OWL-Lite nclos is:

```
(owl-individual! 'Obj1 'Cb)
```

which declares a new individual, '`Obj1`', as a member of the class '`Cb`'.

Properties in the OWL-Lite nclos

This is probably the place where differences with the default nclos are the most important. In the OWL-Lite nclos, properties are first-class ontology elements and they are described and stated outside of class and individual declarations.

A domain of a property limits the individuals to which the property can be applied. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, the property hasChild may be stated to have the domain of Mammal. From this a reasoner can deduce that if Frank hasChild Anna, then Frank must be a Mammal. Note that rdfs:domain is called a global restriction since the restriction is stated on the property and not just on the property when it is associated with a particular class.

The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. For example, the property hasChild may be stated to have the range of Mammal. From this a reasoner can deduce that if Louise is related to Deborah by the hasChild property, (i.e., Deborah is the child of Louise), then Deborah is a Mammal. Range is also a global restriction as is domain above.

Hence the declaration of a property states its name, domain and range.

```
(owl-property! 'p1 'Ca 'Thing)
```

The previous statements declares the property '`p1`', with its domain being all instances of the class '`Ca`' and its range any instance of the root class "Thing". This states that any individual of class "Ca" may have a property "p1" which value is any individual—as all individuals are instances of the root class.

If the property's value is a basic type (number, string) rather than an individual, its range should be defined as '`'Thing`' anyway. The OWL-Lite nclos provides a special construct, called a *scalar*, to handle basic types.

In the current first stage of the implementation of NClosEmacs, all properties ranges are considered scalar.

Slots in the OWL-Lite nclos

Slots in an OWL-Lite ontology are "concrete" properties. A slot, which has an individual id, associates a property with a specific domain individual and a specific range value. For instance,

```
(owl-slot! 'mySlotId 'p1 'Obj1 (owl-scalar! 5))
```

assigns the value 5 to the property '`p1`' of the individual '`Obj1`'.

Ontologies in the OWL-Lite nclos

A complete ontology is then constituted of four lists: classes, individuals, properties and slots. In the OWL-Lite nclos, such an ontology is defined and asserted with the following statement:

```
(nclose-use-nclos ':OWL-LITE
  (onto! 'owl-bench1
 ;; Classes
 (list
  (owl-class! 'Thing nil) ;; Required
  (owl-class! 'Ca 'Thing)
  (owl-class! 'Cc 'Cb)
  (owl-class! 'Cd 'Thing)
  (owl-class! 'Cb 'Ca)
  (owl-class! 'Ce 'Cd)
  (owl-class! 'Cf 'Cd)
  )
 ;; Individuals
 (list
  (owl-individual! 'Obj1 'Cc)
  (owl-individual! 'Obj2 'Cf)
  (owl-individual! 'Obj3 'Cd)
  )
 ;; Properties
 (list
  (owl-property! 'p1 'Ca 'Thing)
```

```
       (owl-property! 'p2 'Ca 'Thing)
       (owl-property! 'p3 'Ca 'Thing)
       )
      ;; Slots
      nil
      )
     )
```

The function '`nclose-use-nclos`' is now used to install an OWL-Lite ontology—rather than the default nclos, in which case its first argument is '`nil`'.

The ontology itself is defined within the '`onto!`' statement by its name followed by the four lists of classes, individual, properties and slots. In this specific example no slots are defined in the ontology. (Slots, however, may be defined dynamically during the inference when, for instance, questions are asked interactively.)

Writing rules for the OWL-Lite nclos

The result of the independence between the rule and the object system in the NCLosEmacs design is that rules are written the same way whether for the default nclos or the OWL-Lite nclos. See [Writing Rules with Patterns in the default nclos], page 11, and See [Writing Rules with RHSes in the Default Nclos], page 14.

The scope of the LHS operators is expanded in the OWL-Lite nclos as it supports inheritance. Hence the '`all-in, some-in, oone-in, none-in`' patterns now select all instances of the mentioned class *and* of all its subclasses recursively.

3.2.4 The Topic Maps Object System in NClosEmacs

In the current version the XTM ontology representation is not yet available.

3.2.5 Examining Objects and Classes

The current implementation provides two interactive commands for printing the current state of the ontology.

nclose-print-instances
: This command prints at the insertion point all individual instances of the selected class, and its descendant subclasses if the currently installed nclos supports inheritance.

nclose-print-object
: This command prints at the insertion point all the properties and their values for the selected object.

Later implementations should provide a proper class/object navigator.

4 Running a session

The first step is to load the NClosEmacs engine (from Emacs) by issuing the command '`M-x load-file nclose.el`'. This basically loads the core engine and initializes it for the current Emacs session.

The preferred way to run an expert session in NClose is to open the rule base text in a new Emacs buffer and evaluate its content with the usual interactive Lisp command '`M-x eval-current-buffer`'. This command compiles the rule base and sets up the rule interpreter for a new session.

A complete expert system session is usually a three-step affair once the appropriate rule base has been loaded. There is an initial volunteering/suggesting phase where the user is expected to either volunteer some initially known data or focus on one or several hypotheses of interest (or both). The rule interpreter is actually started with the interactive *knowcess* command which triggers interactive evaluation of relevant rules. Once this second phase terminates, log traces and various data exploratory commands are provided to investigate answers and results reached by the inference engine. Other commands are available to restart a session or to reinitialize the engine altogether.

4.1 Interactive volunteering and suggesting

Suggesting is the process through which an hypothesis is posted on the agenda for evaluation. The rule interpreter evaluates an hypothesis by collection all rules leading to the selected hypothesis and evaluating them in a backward-chaining mode.

Note that, in contrast to rules engines like RETE, all rules sharing the same hypothesis are evaluated and implicitly or-ed to find the boolean value of the said hypothesis. In the default implementation the order in which these rules are evaluated is left undefined. Of course, the backward-chaining evaluation function could be advised should a specific sequencing of the rules or a lazy evaluation be required for a particular domain application.

nclose-suggest

This function is invoked with '`M-x nclose-suggest`' to suggest an hypothesis. It works with the Emacs completion mechanism so that the usual '`? TAB SPC`' special characters may be used to select the hypothesis to suggest.

Volunteering is the process through with a known piece of data is volunteered to the NClose engine, usually before running an interactive session.

nclose-volunteer

This function is invoked with '`M-x nclose-volunteer`' to volunteer the value of a sign. The user is prompted for the sign first (with standard Emacs completion available) and then for the value.

Several suggest and volunteer commands may be issued before running a session to provide initial information to the expert system.

4.2 Knowcessing restarting and reinitializing sessions

Knowcessing a contraction of *knowledge processing* denotes the process of running an interactive expert system session with NClosEmacs. Once the initial suggest and volunteer phase is completed, the rule interpreter is started with the '`knowcess`' command. User interactions happen in the mini-buffer as the inference process might need to prompt the user for information.

nclose-knowcess
: This interactive command invoked with '`M-x nclose-knowcess`' without argument.

When a question is asked to the user, interactions happen in the mini-buffer; the session may be aborted by typing '`C-g`'.

The expert system session may be restarted at the end of the knowcess phase with the following command:

nclose-reset-session
: This interactive command invoked with '`M-x nclose-reset-session`' restarts the expert system session, setting all signs back to unknown status and emtying the agenda.

nclose-reset-globales
: This interactive command invoked with '`M-x nclose-reset-globales`' without arguments reinitializes the rules engine. In particular, all currently loaded rule bases are deleted from the production memory.

There is also a less frequently used function:

nclose-reset-signs
: This interactive command invoked with '`M-x nclose-reset-signs`' unbinds all previously known signs.

4.3 Encyclopaedia

The current implementation provides only a few functions to pretty-print the list of signs and hypotheses with their values. Gathered under the exaggerated headline of "Encyclopaedia" these functions are as follows:

nclose-print-wm
: This interactive command which is invoked with '`M-x nclose-print-wm`' inserts a table of signs and their values at the cursor position.

nclose-print-hypos
: This interactive command which is invoked with '`M-x nclose-print-wm`' inserts a table of hypotheses and their values at the cursor position.

4.4 Session logs

A complete session log is maintained in a special Emacs buffer named '`*nclose-log*`'. This buffer keeps track of the inference progress from the initial knowcess command. The buffer is erased with each reset session command issued by the user.

Note that logging is implemented as a specific aspect layer in the '`advice.el`' file, See Section 2.2 [Aspect Oriented Programming], page 5. It is then easy to switch logging on or off by simply enabling or disabling this particular layer in the file. In addition, more detailed levels of trace are simply implemented by creating new advising functions for events of interest in this layer.

5 Implementation

This section describes in more detail some of the features in the current implementation of NClose basic operations.

The implementation of NClosEmacs relies on the underlying ELisp interpreter in Emacs. The primary design choice was to represent each sign and hypothesis in rules as Lisp symbols, the value of which would be the value referred to or inferred by the expert system.

As a consequence of this choice, rules are Lisp macro forms which call internal functions managing a few global (to the ELisp environment) variables used by the rule interpreter. Note that as signs are Lisp variables, they are available in the ELisp environment available with each Emacs session. In other words, the inference engine and the loaded rule base(s) are indeed available whatever the buffer displayed in Emacs provided they were initialized and loaded in the same session.

As a reminder, NClosEmacs is made available to the current Emacs session by loading the '`nclose.el`' file with the standard command '`M-x load-file`'; a rule base is loaded for interpretation by the NClosEmacs engine by opening a buffer on the rule base file ('`C-x C-f`' or '`C-x C-v`', for instance) and evaluating the buffer with the standard interactive command '`M-x eval-current-buffer`' in Emacs.

The interpreter relies on a global agenda, a first-in first-out stack of hypotheses to evaluate. Gating, which is enabled by default, is responsible for posting hypotheses on the agenda as the inference progresses. (Further research might in fact unveil other inference coprocesses that would interact with the agenda during an expert system session.) The essential interpreter process pops out the next hypothesis from the agenda, stopping it it is empty, and passes it to the evaluator.

The evaluator implements a simple backward-chaining mode. Rules leading to the hypothesis are first collected from production memory. All are in turn evaluated by attempting to pass their LHSes to the ELisp evaluator (the '`eval`' function). On a failure to evaluate due to an unbound symbol, which in the chosen representation stands for an unknown sign, a question is asked to the user and evaluation is retried until it is successful. Individual rules evaluations are implicitly or-ed to assign a boolean value to the hypothesis, triggering the corresponding RHSes if present.

As previously mentioned, this implementation is also intended as a base for further research. A promising direction is to look into advising each of the previous evaluation steps to evolve domain-specific inference engines for vertical applications, while keeping the general architecture and rules syntax of NClosEmacs. Another direction of research is to investigate attachment of other cognitive coprocesses to the inference itself, again using advising functions in system-specific layers. (Machine Learning and DDB/TMS being prime candidates for inference coprocesses in the NCLose architecture.)

Finally NCloseEmacs runs in the Emacs universe. It is only natural that it should come with a rudimentary, at this stage, major mode for authoring and running rule bases. The current implementation, which uses the generic mode approach, is only a first look into that development.

5.1 Rules compilation

An introductory word about the basic data structures in this implementation of NClosEmacs.

In this version, the four global variables required for the interpreter are simply implemented as lists.

- nclose-global-signs is the list of signs collected by "compiling" the rules
- nclose-global-hypos is the list of hypotheses kept in a separate list; note that an hypothesis might be a sign if it is mentioned in another rule's LHS (the '`Yes`' and '`No`' operators operate on hypotheses as well as on boolean signs).
- nclose-global-pm is the list of rules currently loaded
- nclose-global-agenda is the agenda, also implemented as list.

Other global variables are possibly added by advising functions as required for the operation of their layer. This is in fact the case for the gating and logging layers.

ELisp offers several alternatives to list for the implementation of the global interpreter variables: hash-lists, sequences, property lists and association lists are valid candidates for alternative implementations. Furthermore, several extension packages exist for ELisp and Emacs to integrate with external repositories such as relational databases, file systems or remote servers. These could be explored in further research work.

5.2 Agenda Driven Evaluation

The agenda keeps track of hypotheses to evaluate. The initial hypotheses are posted through the suggest command and, indirectly, by the initial volunteer command. Other hypotheses may be posted during the inference process.

However, when the evaluation of a rule finds a yet unbound hypothesis, it starts evaluating that hypothesis at once, without posting it to the agenda.

5.3 Classes and Objects

This section is a first draft. It covers the basics of the default and the OWL-Lite nclos.

5.3.1 Introducing ontologies in NClosEmacs

Objects and classes were introduced in Nexpert Object as early as 1986. The design of the original class and object system reflected the early thinking on object systems which was barely emerging at the times. Partly inspired from object-oriented programming languages such as Samlltalk and C++, partly from frame-based and AI research object systems, it distinctly had a set-theoretic flavor. Patterns expressed subset selections from classes or subobject selections from objects following a classical element/set distinction. On the other hand, Nexpert Object supported a fairly rich inheritance system with overriding, pre- and post- method executions, exception handling and dynamic object attachment/detachment.

In the current NClosEmacs incarnation, we chose to leverage progresses made since then in formalizing the notion of *ontology* which now sits at the core of the Semantic Web iniative. More specifically a nclos is based on a particular specification of an ontology. The default nclos, for instance, has a simple ontology specification : classes are considered as sets of objects, each having a series of scalar properties, with no inheritance and no complex property types. The OWL-Lite nclos, on the other hand has a much more comprehensive class, object and property representation, with inheritance and complex types —although currently the LHS pattern language and the RHS commands have not yet been extended to benefit from the enriched types. The common ground for the formal definition of ontologies in NClosEmacs is a structure made of:

- a list of classes each uniquely identified by a name or id, each class may be attached to a parent class or to the root class (by convention '`Thing`').
- a list of objects each uniquely identified by a name or id and attached to a parent class.
- a list of properties, also uniquely identified by a name or an id, each with a scope—which is the class which instances it attaches to—and a range—which is the "type", a class name which instances the property's value is permitted to take.
- a list of slots, each one attaching a value to a property for a given object.

In addition each ontology has a unique name. (In the current implementation only one such ontology may be installed. Future releases may offer better ontology packages handling.)

This simple ontology specification is provided in the '`ontology.el`' file. This file contains the basic '`onto-defstruct`' ELisp macro which can be used by ontology models to instantiates the class, object, property and slot persistent records they need.

For instance, the OWL-Lite nclos defines its classes as:

```
(onto-defstruct owl-class ((:finder . id)) id subClassOf)
```

which specifies that a class in the OWL-Lite nclos has an id, which is unique and searchable for in the persistent store, and a '`subClassOf`' which contains the id of the parent class. With this ontology specification—in this case defined as expected in the '`owl-lite.el`' file—concrete OWL-Lite classes in a knowledge base are declared with ELisp forms like:

```
(owl-class! 'Thing nil)    ;; This is required
(owl-class! 'Winery 'Thing)
(owl-class! 'Region 'Thing)
(owl-class! 'ConsumableThing 'Thing)
(owl-class! 'Glen 'Winery)
```

These declarations have side-effects and create automatically getters and setters, testers and finders for their classes:

owl-class? Returns true if its argument is an '`owl-class!`' form.

owl-class-id
Getter for the '`id`' name property of the class argument.

update-owl-class-id
Setter for the '`id`' name property of the class argument. It is also associated to setf, so that the invocation '`(setf (owl-class-id class) 'New-Name)`' works as expected.

owl-class-subClassOf
Similar definition for setter as above.

update-owl-class-subClassOf
Similar definition for setter as above.

onto-find-owl-class id list-of-classes
This function performs a basic search on the '`id`' of a class, returning the '`(owl-class! ...)`' that matches, if any, in the list passed as the second argument.

These functions are used in the implementation of the adaptation of the pattern language to the nclos in point. They may be used by programmers to extend the ontology services in NClosEmacs as well.

All elements in an ontology model use the '`onto-defstruct`' construct for their basic building blocks. See Section 5.4 [Adding new object systems], page 25

5.3.2 Persistence and Volume

Issues of persistence and volumes are not addressed in the current implementation of nclos models. Complete ontologies are kept in memory as ELisp lists.

5.3.3 Variables versus Slots

In the current implementation they are two types of scalar—basic type—variables. System-wide variables such as '`X`' in the following LHS fragement:

```
(> X 3)
```

and slots, the value of which is also of a basic type, such as '`p3`' in:

```
(some-in 'Class-A 2 (> p3 3))
```

In the original Nexpert Object, system-wide variable were automatically updated to object type with special property '`value`', so that the first condition would read:

```
(> (prop-in 'X 'value) 3)
```

Another option with the current implementation would be to consider system-wide variables as local slots of a '`System`' object, so that the former LHS fragment would in fact read:

```
(> (prop-in 'System 'X) 3)
```

with the same effect.

The current implementation maintains separate ELisp variables for each at this stage. This precludes, for now, the use of system-wide variables within the third argument expression of a pattern operator such as all-in, some-in, and so forth.

5.3.4 Overview of the nclos API

The articulation between the rule engine and the ontology management is defined in the '`nclos.el`' file which specifies a set of ontology related functions required by the engine for evaluation, selection and execution of rules. The most important are:

nclos-slot-reader sym prop
nclos-slot-writer sym prop value
: These are the getter and setter for a slot value, which is identified by its object and its property.

nclos-slot-reset
: This function reinitializes the list of slots to the original ontology definitions installed in the knowledge base.

nclos-find-instances class
: Retrieves a list of ids of all the instances of the passed class (recursively if inheritance is supported in the current nclos).

nclos-find slots obj
: Enumerates conses of property id and value for slots attached to the passed object. This is mostly used for tracing and logging purposes.

nclos-find-all-classes
nclos-find-all-objects
: Returns as a list the store of classes or objects, accordingly.

When extending NClosEmacs with a new nclos model, these functions, and other ancillary functions in the same file, have to be extended to accommodate the new model. This can be done either by adding proper code into the '`(cond ...)`' forms or by using advising functions. In both cases the '`globales.el`' and the '`reset.el`' might also require some alterations.

Usually the extensions of these functions make use of separate model-specific files as is the case for instance for:

- Default nclos model. Refers to the '`set-unification.el`' and '`set-instances.el`' files.
- OWL-LITE nclos model. Refers to the '`owl-lite.el`' and '`owl-instances.el`' files which rely on the '`ontology.el`' base.
- EIEIO nclos model. Refers to the '`nclos-eieio.el`' file. See Section 5.4 [Adding new object systems], page 25

In contrast to the current (redundant) implementation, this separation would allow to on-demand loading of the required code for a given nclos model at installation time. This is not yet the case in this first pass implementation where all models are indeed available all the time.

5.4 Adding new object systems

This section explains how to add a new object system to NClosEmacs. More specifically, this section details how the EIEIO object system, part of the CEDET suite of tools for Emacs, may be added to NClosEmacs.

5.4.1 Overview of classes and objects

This section highlights the features of classes and objects in EIEIO which are relevant to the NClosEmacs rule system.

The class and object system in EIEIO draws its inspiration from the CLOS standard in the early eighties. Its architecture is different from the modern “ontology” approach of the Semantic Web, best exemplified in RDF and OWL. Along the lines of the reference *The Art of the MetaObject Protocol*, EIEIO provides classes, objects, generic functions and methods. The integration path discussed in this chapter is only one way of articulating a NClos based on the ideas in *The Art* but would basically work for other CLOS-inspired ELisp implementations. (See also Closette.)

The integration proposed here is implemented in the ‘`nclos-eieio.el`’ file and supposes the EIEIO library to be properly installed in the load-path of Emacs. Its basic design options are:

- Directly use classes and objects defined in EIEIO as classes and objects in the NClose set-oriented perspective.
- Similarly to the OWL implementation, define an “ontology” abstraction; here simply to bundle classes and objects effectively used in a rule base in a namespace.
- Map as directly as possible EIEIO-provided inheritance mechanisms to NClose inheritance requirements.

These options simplify the integration work and reduce the amount of code necessary to implement the NClos API to EIEIO.

5.4.2 NClos class implementation

The minor departure from classes and objects as required in the NClose inference engine and CLOS-inspired class/object system is the set oriented perspective imposed by patterns in rules LHS (and set commands in the RHSes). Operators like ‘some-in’ or ‘all-in’ trigger evaluation on all instances of said class as their side-effect is to build the list of matching objects to be passed to the rule RHS. The classes used in the rule system hence are required to keep track of their instances. Fortunately, this is easily implemented in EIEIO via the use of the special class ‘`eieio-instance-tracker`’, defined in the extension ‘`eieio-base`’ module.

Here we chose to define a new keyword ‘`nclose-defclass`’ to supersede the standard CLOS ‘`defclass`’ form for those classes which are used in a NClose knowledge base. This specific class definition form simply attach the defined class to the root ‘`eieio-instance-tracker`’ class rather than the default one. Its definition is simply:

```
(defmacro nclose-defclass (name superclass fields &rest options-and-doc)
  "Defines a NClose class for use in the knowledge base.  These classes are otherwise r
  ‘(eval-and-compile
     (defvar ,(util-s-cat name "instances") nil)
     (eieio-defclass ’,name
     ;; Attach to a different metaclass
     ’,(append superclass (list ’eieio-instance-tracker))
     ;; Add the required init forms for metaclass-inherited
```

```
          ',(append fields
            (list
     (list 'tracking-symbol
           ':initform (util-s-cat name "instances"))))
          ',options-and-doc)
            )
       )
```

Note that this form defines a global variable, called ‘`class-instances`’, for each new class in order to track its instances as they are created and deleted. This global variable is kept as the value of the ‘`tracking-symbol`’ slot for each class.

With this convention, the set of instances of a given class is simply:

```
     (defun nclos-eieio-instances (class)
       "Returns a list of the class instances, one level deep."
       (symbol-value (util-s-cat class "instances")))
```

The methods in class ‘`eieio-instance-tracker`’ are responsible for updating the tracking symbol at each instance creation of deletion.

Finally the ontology abstration is a class defined as:

```
     ;; Experimental representation syntax, inspired loosely from OWL-Lite.
     (defclass eieio-ontology ()
       ((onto-id
         :initarg :id
         :documentation "The unique name of the NClose class."
         )
        (onto-classes
         :initarg :classes
         :documentation "The list of CLOS/EIEIO class names in the NClose namespace."
         )
        (onto-objects
         :initarg :individuals
         :documentation "The list of CLOS/EIEIO object names in the Nclose namespace."
         )
        )
       :documentation "The ontology (super)class."
     )
```

This simplistic class keeps tracks of all classes and objects relating to a knowledge base. It could be enriched and extended towards a full namespace management with ontology-level primitives and commands, for instance.

Note that as NClose classes are indeed EIEIO classes, they may be extended with generic functions and methods as for any other standard class.

5.4.3 NClos object and slot implementation

With the class implementation in place, NClose objects are none other than EIEIO objects, instances of '`nclose-defclass`' defined classes. They are defined and used as would be any EIEIO standard object. As an example, here is a complete ontology form as declared in a knowledge base file:

```
(nclose-use-nclos
 ':EIEIO
 (eieio-ontology "Test ontology"
  :id "Experimental ontology"
  :classes (list
    (nclose-defclass Ca ()
     ((pressure :initarg :pressure
:documentation "Measure at T0")
      (temperature :initarg :temperature
    :documentaion "Heat Excess")
     (volume :initarg :volume))
    "Tank class")
    (nclose-defclass Cb (Ca)
    ((manufacturer :initarg :manufct)
     )
    "Subtank with identified manufacturing origin")
    (nclose-defclass Cc (Ca)
    ((duration :initarg :duration)
     )
    "Subtank with identified alert duration")
   )
 :individuals (list
       (Ca "O1-name" :pressure 200 :temperature 100 :volume 10)
       (Ca "O2-name" :pressure 200 :temperature 200 :volume 20)
       (Ca "O3-name" :pressure 200 :temperature 300 :volume 30)
       (Cb "O4-subtank"
                   :manufct "GE"
                   :pressure 200 :temperature 100 :volume 10)
       (Cc "O5")
       )
 )
)
```

This fragment shows an ontology with 3 classes, the first one being parent to the next two, and 5 objects variously defined.

Note that, in contrast to Semantic Web inspired ontologies, slots are not separately defined as first-class properties like in OWL-LITE. Here the slots are implicitly defined in the class specificaiton and valued in the object declarations themselves.

5.4.4 Query functions

The get/set and query functions are the core of the NClos API implementation. Adding a new object system involves adding dispatch calls in the '`nclos.el`' file to issue appropriate getter, setter and query calls according to the installed nclos. (Which means, in particular, that for a given knowledge base there is only one currently installed nclos, namely the one resulting from the last '`nclose-use-nclos`' call.)

For instance, the slot reader should look like:

```
(defun nclos-slot-reader (sym prop)
  "Get value of the slot identified by sym.prop, which may be unbound."
  (cond
   ((equal ':OWL-LITE nclose-global-nclos)
    (owl-property-reader sym prop)  ;; See 'owl-instances.el'
    )
   ((equal ':EIEIO nclose-global-nclos)
    (eieio-property-reader sym prop) ;; See 'nclos-eieio.el'
    )
   (t
    (let ((val (get sym prop)))
      (if val val
(read-minibuffer (format "What is the value of the %s of %s? " prop sym)))))
   )
)
```

and similarly for the other dispatch functions.

Reading and Writing Slot Values

With the previous design options in place, the getter and setter code for EIEIO is as follows:

```
(defun eieio-property-reader (sym prop)
  "Reads in the scalar value of 'prop' in object 'sym' in the current EIEIO ontology."
  (if (slot-boundp sym prop)
      (slot-value sym prop)
    (read-minibuffer (format "What is the value of %s of %s? " prop (object-name sym))
    )
  )

(defun eieio-property-writer (sym prop val)
  "Writes value 'val' to 'prop' in object 'sym'."
  (set-slot-value sym prop val)
  )
```

Note that these functions may be advised, if required, to implement additional inference mechanisms such as gating or context associations on individual slots or objects. (In the current implementation, though, it is not the case.)

Querying the Ontology

Query functions are required for the encyclopedia commands to output class and object descriptions. They are also useful to implement browsers and other visual representation of the knowledge base. The basic query functions defined in the NClos API are:

- '`nclos-scope-acc`' used in LHS unification to create a list of all instances of a given list of classes
- '`nclos-find-instances`' to provide a list of all objects in passed class argument
- '`nclos-find-all-classes`' to list all NClose defined classes in the knowledge base
- '`nclos-find-all-objects`' to list all NClose objects in the knowledge base, independently of their class attachment
- '`nclos-find-slots`' which returns a list of consed slot value pairs for a given object (or object name) argument.

As NClose inheritance abstraction specifies that instances of subclasses are also instances of their superclasses, the firs step is to implement a recursive exploration of the class inheritance tree to produce all instances of a given class. This is what the '`nclos-eieieo-all-instances`' function does in a NClos-dependent way.

An example implementation of the unification initialization primitive for EIEIO classes:

```
;; Tail recursive implementation of initial extensions environment
(defun eieio-scope-acc (specs env)
  "Initializes the initial extensions for rules evaluation in EIEIO."
  (cond
   ((null specs) env)
   (t
    (let ((ext (map-append
(nclos-eieio-all-instances (extract-class (car specs)))))
    )
(put env (extract-class (car specs)) (make-vector (cdar specs) ext)))
    (eieio-scope-acc (cdr specs) env)
    )
   )
  )
```

The result is provided as the symbol plist for 'env' which lists, for each class mentioned in a rule LHS, the initial list of its instances, called the *scope*. As unification proceeds, the scope is further reduced by filtering it out according to the LHS patterns. The final scopes are passed to the RHS, when all conditions are matched.

The remaining query functions are somewhat simpler as they rely on calling to the global ontology currently installed, an instance of the '`eieieo-ontology`' class.

```
(defun eieio-find-all-classes (onto)
  "List all class symbols defined in current ontology."
  (mapcar '(lambda (class-spec) (elt class-spec 1)) (slot-value onto :classes))
  )
```

```
(defun eieio-find-all-objects (onto)
  "List all individual names (objects) defined in ontology."
  (mapcar 'object-name (slot-value onto :individuals))
  )
```

Finding instances of a class is done as follows:

```
(defun eieio-find-instances (class)
  "List all instances (recursively) of an EIEIO class. `class' is a symbol."
  (mapcar 'object-name (map-append (nclos-eieio-all-instances class)))
  )
```

Note that the string-typed names of objects are returned here rather than their raw vector symbol representation in EIEIO.

Similarly for slots of a given object:

```
(defun eieio-find-slots (objname)
  "List conses of prop . values for slots of `obj'"
  (let ((obj (find-object objname nclose-global-ontology))
)
    (mapcar '(lambda (x) (cons x (slot-value obj x))) (object-slots obj))
    )
  )
```

Note that the argument here is the name of the object, as a string, rather than the object symbol itself, hence the '`find-object`' query against the ontology.

5.4.5 Packaging for the NClosEmacs library

The suggested packaging for adding a new nclos system to NClosEmacs is to use the 'provide/require' facilities for ELisp. The NClos API implementation would usually '`require`' the class/object library and '`(provide 'nclos-class-object-library)`' to NClosEmacs. The following steps are usually required:

- Implement the NClos API for the selected class/object library and use provide/require
- Edit the mamefile to trigger compilation of the previous implementation
- Edit '`nclos.el`' to appropriately dispatch calls to your library
- If required, add specific keywords and commands to the NClose major mode in '`nclose-mode.el`'
- Finally make sure the class/object library is in the load-path specified in the 'makefile', with the make variable '`LOADPATH`'

The library is rebuilt by issuing the '`make release`' command within the NClosEmacs directory.

5.4.6 On the EIEIO nclos

The 'SATFAULT' directory in the release contains a revived antique example of knowledge base which was consistently used in demonstrations. The 'SATFAULT' knowledge base is provided in two forms, one with an OWL-LITE ontology, the other with an EIEIO implementation of the same ontology.

In this EIEIO experimental implementation, the release benefits from the tree visualization in the '`tree.el`' file. This EIEIO-specific features provides a basis for a simple *ontology browser* implementation. This simple browser is invoked through the interactive command 'nclose-browse-class'.

For a given class, the browser shows the tree of its subclasses. With the cursor positioned on the name of a class in the browser, several keyboard commands become available:

- 'e' shows the documentation for the class, including its slots;
- 'x' lists the instances of the class

(Additional commands will become abailable in further releases.)

5.5 Aspects

As a practical research matter, this implementation looks at aspects to support some of the NClose behaviors. In the current version of NClosEmacs, AOP has been investigated to develop so-called inference *coprocesses*, additional computing processes that occur while inference is progressing. The coprocesses we have in mind range from simple logging of user actions and inference engine deductions to complex machine learning algorithms or truth maintenance systems. In the current release we have experimented with using aspects, or to be more specific context-oriented programming, for the NClose gating mechanism and for logging.

5.5.1 The Gating Aspect Layer

Gating is somewhat typical of an inference coprocess in the previously proposed sense. A generic inference coprocess is expected to rely on some data structure, possibly dependent on the rule base, which hence requires initialization and re-initialization with the expert system sessions. At inference time, the coprocess data structure is likely to be used and modified as inference progresses.

In the case of gating in Nclose, a record of associations between signs and hypotheses is collected when rules are loaded and compiled in memory. This forward association information is implemented here as a simple global property list. As previously mentioned, ELisp offers many alternatives for consideration as well. At inference time, whenever a sign becomes known its associated hypotheses are posted (in no particular order) on the agenda for later evaluation.

Since the aspect layers are all implemented through ELisp advising functions, the later are then required at initialization and reset time to create and set up the gating data structure, i.e. the forward association property list; at load-and-compile time when rules' LHSes are parsed; and, of course, at run time when the inference process assigns values to unknown signs. This is a typical advice layer architecture as provided in the '`advice.el`' file.

In particular the NClosEmacs functions that require advising functions in this layer are:

- nclose-global-init
- nclose-reset-globales

to synchronize the gating data structure and sessions,

- sign-compile

to capture associations between LHS signs and hypotheses at load-and-compile time,

- sign-writer

to cause associated goals to be posted on the agenda when signs become known at inference time.

Using AOP gating is implemented with four advising functions at minimal cost.

5.5.2 The Logging Aspect Layer

This layer exemplifies a cross-cutting concern where a much larger number of functions in the implementation are concerned. Depending on the level of details required for logging a few, some or many functions need to be advised in this layer.

The current implementation logs these events:

- start of the inference process (*knowcess*)
- beginning of the evaluation of a new hypothesis
- beginning of the unification of a rule's LHS
- beginning of the firing of a rule's RHS
- assignment of a value to a sign

Of course, adding more detailed tracing facilities is a matter of complementing these with new advising functions for events of interest.

All events are logged into a special buffer named '`*nclose-log*`' and available through the usual buffer commands in Emacs. Note that this buffer is emptied at each session reset and should be saved if required for later inspection.

5.6 The NClose major mode

The NClose major mode, still pretty much work in progress at this stage, is part of the default front-end to NClosEmacs. The general idea is that rule bases are simple text files edited in standard Emacs buffers. All major interactions with the expert system are designed to happen from within these buffers. Hence all user commands are interactive Lisp functions usually invoked with the '`M-x`' prefix. Of course, some NClosEmacs specific buffers, such as the log buffer, might also be involved, or commands such as the Encyclopaedia may also work in any buffer, but the main idea is that major user interactions are centered around the rule base buffer itself. (Note: this might evolve in later versions when classes/objects are added.)

The NClose major mode, in its current implementation, serves as a blueprint for a comprehensive expert system dashboard. In order to start from a simple base, this mode is, for now, derived from the Emacs generic mode so as to experiment with various design options. Its major function in this revision is to font-lock hypotheses in the rule base text to reflect the progress of the inference process.

```
(define-generic-mode 'nclose-mode
  '(";")
  '("add-to-kb")
  '(("(@hypo[ ]+\\([a-zA-Z0-9\-]+\\))" 0 (hypo-facespec (match-string 1)))

     )
  '()
  '()
  "NClose-mode is a major mode for authoring and knowcessing rule bases."
)
```

As can be seen from the code, the major mode highlights the rule definition keyword, and parses the rules source for hypothesis, selecting a font according to its status (under evaluation, known true or known false).

Nclose mode is invoked with '`M-x nclose-mode`' when in a rule base text buffer. It is also, in Emacs fashion, automatically set for a buffer if the first line of the file declares:

```
;; -*- mode: nclose -*-
```

a standard way to tell Emacs to switch to the appropriate major mode when displaying this file.

6 A Conclusion and a Starting Point

The NClosEmacs implementation is both a valid and complete implementation of the NClose inference engine, as originally thought out more than twenty five years ago, and a new base for further research work.

New design choices, more specifically the leverage of the AOP and context-oriented programming approach, which are investigated in this implementation, open new avenues of research. The emergence, in particular, of inference coprocesses as a critical formulation of extra features of the inference engine with the added benefit of an explicit implementation technique through advising functions is one of the remarkable facet of NClosEmacs.