ConceptBase V3.1 User Manual

Matthias Jarke, Ed.[1]

ConceptBase is an experimental deductive object base management system specifically intended for the coordination of design environments. It integrates techniques from deductive, object-oriented, and temporal databases in the logical framework of the data model Telos. The usage environment offers an extensible palette of graphical, tabular, and textual windows conforming to the X11 standard. The communication between the windows and the object base is organized in a client-server architecture using the wide area network protocol Internet.

[1] Contributions to this manual were made by: Markus Baumeister, Stefan Eherer, Rainer Gallersdörfer, Manfred Jeusfeld, Christof Lenzen, Hans Nissen, Martin Staudt, and Klaus Weidenhaupt.

Contents

1 Introduction

ConceptBase is a deductive object base management system. Its data model is a conceptual modeling language making it particularily well-suited for design applications. Earlier prototypes [JJR88, EJJ*89] have been used in projects ranging from development support for data-intensive applications [JMSV90], requirements engineering [RD92], and version&configuration management [RJG*91] to co-authoring of technical documents [HJEK90].

The key features distinguishing ConceptBase from other extended DBMS and expert systems shells are:

- clean formal integration of deductive and object-oriented abstraction
- client-server architecture with wide-area Internet access
- equivalent logical, semantic network, and text frame representations

ConceptBase implements a version of the knowledge representation language Telos [MBJK90]. Telos integrates a thoroughly axiomatized structurally object-oriented kernel with a predicative assertion language in the style of deductive databases. A complete formal definition can be found in [JEUS92]. Telos is viewed as a hybrid knowledge representation language integrating ideas from frame-like concept description languages and deductive databases. This hybrid character suggests a hypertext-style combination of (graphical) semantic network views and (textual) frame representations at the user interface. Both Telos views are based on the common logical representation so that they can be treated in a completely symmetrical way, both for purposes of querying/browsing and of editing/ manipulation.

1.1 ConceptBase Architecture

ConceptBase V3.1 follows a client-server architecture. Clients and servers run as independent processes which interact via inter-process communication (IPC) channels (Fig. 1-1). Although this communication channel was initially meant for use in local area networks, it has been used successfully for nationwide and even transatlantic collaboration of clients on a common server.

The *ConceptBase* server offers a programmer interface which responds to messages expressed in Telos syntax. In V3.1, this interaction emphasizes the role of so-called query classes which can be understood as parameterized views; various rewriting-based optimization techniques from deductive databases have been extended to our object-oriented setting and are included in the query evaluation and integrity checking subsystems [JJ91,JS92].

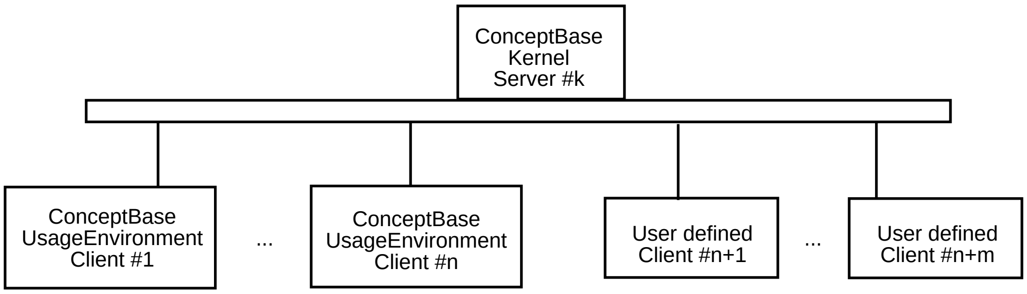


Fig. 1-1: The client-server architecture of *ConceptBase*

A configuration of standard clients, the *ConceptBase* usage environment, supports a number of window-based tools, some of them Telos-generic, some supporting specific design models. The X11-based environment includes a tool for co-authoring of hypermedia documents. Additional client programs can be developed by the user; they are attached to the communication channel by an IPC interface. More than one client may be connected to the same server process, but *ConceptBase* does not yet support concurrency control beyond serialization of messages.

1.2 Hardware and Software Requirements

ConceptBase operates on SUN-4 workstations (and clones) under SUN-Unix 4.1.1. The main implementation language is ProLog by BIM V3.1 [BIM90], using interfaces to C for specialized procedures. The X11-based usage environment is entirely written in C, using the Andrew toolkit. Therefore, it can be ported to any Unix workstation with compatible X11 and Andrew toolkit versions. Persistence and recovery are achieved through a file system. A machine with at least 16 MB of main memory is recommended for efficiency, other system requirements are mentioned in the Installation Guides (appendix 1). *ConceptBase* is distributed in one of the following ways: as an interpreter version if a full development license for ProLog by BIM is available or as a runtime version if no such license exists[1] .

1.3 Overview of this Manual

This manual provides information about installing and using *ConceptBase*. More information about the knowledge representation mechanisms, the applications, and the implementation concepts can be found in the references.

Section 2 describes the *ConceptBase* version of the Telos language and gives some examples for its usage. Section 3 describes the X11 *ConceptBase* Usage Enviroment. The Telos-based programmer interface of the server, the IPC mechanism of using *ConceptBase* kernel servers, and the methods for attaching user-defined clients are presented in section 4[2] .

---

[1] We would like to thank BIM for the possibility of distributing also a runtime version of our system

[2] A detailed programmer's manual can be obtained from Lehrstuhl Informatik V, RWTH Aachen, Ahornstr. 55, 5100 Aachen, Germany. Please ask for internal report I5-9203.

Appendices contain an installation guide (A1), a formal definition of the Telos syntax and internal data structures (A2), and the full Telos notation of an example model and a case study on the modeling of entity-relationship diagrams with Telos (A3). A major application of ConceptBase, a co-authoring system in a hypertext environment, is described in appendix 4. Up-to-date information on these issues can also be found in the file *README.1st*. A list of known bugs and limitations can be found in the file *CBBUGS*.

1.4 Differences to Earlier Versions

ConceptBase V3.1 is an intermediate version. The time calculus ist still included in the system but no longer described in the manual. We plan to reimplement it in the next release in an way independent from the data model and the kernel system. The predicates of the logical language are now much more readable. The old SunView-based usage environment is no longer supported due to a decision made by the manufacturer of the operating system.

We tried hard to guarantee maximum compatibility to older versions of *ConceptBase*. For example, the old predicates of the logical language (including temporal parameters) are still accepted. Furthermore, the methods of the server that were available in V3.0 but not documented in this manual will usually work. However, we strongly recommend not to use these undocumented features for future applications.

1.5 Perspective

The goal of the next full release is to provide a much smaller kernel system (measured in number of predicates and methods) but with the opportunity to attach extensions like temporal or spatial calculi, team coordination mechanisms etc. in a simple and transparent manner. Other plans include:

1. Optimized support of logical formulas at the meta class level. This feature makes most properties of data models (inheritance patterns, attribute categories etc.) user-definable. As a consequence, ConceptBase can be used as a system to simulate a broad class of specialized data models.
2. Composition of modular objects.Query classes are extended with a concept of interfaces (public properties) and implementations (private properties, components). This extension supports applications such as development in the large, security, or model-based editing.
3. View update and maintenance.. Modifications to the object base will be possible in terms of derived information, i.e., query classes.
4. Automatic classification as in KL-One-like languages will be integrated.

abcd

2 The Telos Language in ConceptBase

Telos' structurally object-oriented framework generalizes earlier data models and knowledge representation formalisms, such as entity-relationship diagrams or semantic networks, and integrates them with predicative assertions and temporal information. This combination of features seems to be particularly useful in software information applications such as requirements modeling and software process control. A formal description of Telos can be found in [MBJK90, JEUS92]. The following example is used throughout this section to illustrate the language:

> *A company has* `employees`*, some of them being* `managers`*. Employees have a* `name` *and a* `salary` *which may change from time to time. They are assigned to* `departments` *which are* `headed` *by managers. The* `boss` *of an employee can be derived from his department and the manager of that department. No employee is allowed to earn more money than his boss.*

This section is organized as follows: first, the frame and network representation of the Telos language is explained. Then, the predicative sublanguage for deductive rules and integrity constraints are presented. Subsection 2.3 presents a declarative query language which introduces queries as classes with optional predicative membership specification.

2.1 Frame and Network Representation

As a hybrid language Telos supports three different representation formats: a logical, a graphical (semantic network) and a frame representation. The two latter formats are based on the logical one. As explained in the next subsections the logical representation also forms the base for integrating a predicative assertion language for deductive rules, queries, and integrity constraints into the frame representation. We start with the logical so called *proposition* representation of a Telos knowledge base.

A Telos knowledge base is a finite set of interrelated *propositions (=objects)*:

```
KB = {proposition(oid,x,l,y,t,tt) | oid,x,y,t,tt in ID,
                                      l in LABEL}
```

where `oid` has key property within the knowledge base, `ID` is a non-empty set of identifiers with a non-empty subset `LABEL` of names. The components `oid`, `x`, `l`, `y`, `t`, `tt` are called *identifier*, *source*, *label (*or *name)*, *destination*, *validity time* and *belief time* of the proposition. We read them as follows:

> The object `x` has a relationship called `l` to the object `y`. This relationship holds for the time `t` and is believed by the system for the time `tt`.

As shown below there is a natural interpretation of a set of propositions as a directed graph (semantic network). We distinguish four patterns of propositions and give them the following names:

- *Individuals*
  `proposition(oid,oid,l,oid,t,tt)`
  ("`oid` is an object with name `l` at time `t` believed `tt`")
- *InstanceOf* relationships (instantiations)
  `proposition(oid,x,*instanceof,y,t,tt)`
  ("`x` is an instance of class `y` at time `t` believed `tt`")
- *IsA* relationships (specializations)
  `proposition(oid,x,*isa,y,t,tt)` and
  ("x is a specialization of `y` at time `t` believed `tt`")
- *Attributes* (all other propositions)

Telos imposes some structural axioms on knowledge bases, e.g. referential integrity, correct instantiation and inheritance ([JEUS92]). They are linked to predefined classes that are automatically part of each Telos KB:

`Individual` contains all individuals as instances.
`InstanceOf` contains all instantiations as instances.
`IsA` contains all specializations as instances.
`Attribute` contains all attributes as instances.
`Proposition` contains all propositions as instances.
`Class` contains all "full classes" (including itself) as instances: such classes may define deductive rules or integrity constraints.
`Token` contains those individuals that may never have instances themselves.
`SimpleClass` (alias `S_Class`) contains individuals that may have instances which are tokens.
`MetaClass` (`M1_Class`) contains individuals that may have simple classes as instances.
`MetametaClass` (`M2_Class`) contains individuals that may have meta-classes as instances.

Additionally, there are the builtin-classes `Integer`, `Real` and `String`. Integer and real numbers are written in the usual way, strings are character sequences, e.g. `"this is a string"`.

As a user, you don't work directly with propositions but with textual (frame) and graphical (semantic networks) views on them. Both are not based on the oid's of objects but on their label components. To guarantee a unique mapping we need the following naming axiom:

Naming axiom

The label ("name") of an individual object must be unique. The label of an attribute must be unique within all attributes with a common source object.

The **frame syntax** of Telos groups the labels of propositions with common source `o` around the label of `o`. The exact syntax is given in appendix A2.1. In this section we introduce it by modeling the employee example:

```
Individual Employee in Class,SimpleClass with
      attribute
            name: String;
            salary: Integer;
            dept: Department;
            boss: Manager
end Employee
```

```
Individual Manager in Class,SimpleClass isA Employee end

Individual Department in Class,SimpleClass with
      attribute
            head: Manager
end Department
```

The label of the "common source" in the first frame is `Employee`. It is declared as instance of the classes `Individual,Class` and `SimpleClass` and has four attributes. The class `Manager` is a subclass of `Employee`.

Oid's (preceded by '#' in our examples) are generated by the system. The default validity time for objects is `Always` (see remarks below). This leads to the following set of propositions corresponding to the frames above. The belief time inserted by the system is denoted by omission marks.

```
proposition(#Employee,#Employee,Employee,#Employee,Always,...)
proposition(#1,#Employee,*instanceof,#Class,Always,...)
proposition(#2,#Employee,*instanceof,#SimpleClass,Always,...)
proposition(#3,#Employee,name,#String,Always,...)
proposition(#4,#Employee,salary,#Integer,Always,...)
proposition(#5,#Employee,dept,#Department,Always,...)
proposition(#6,#Employee,boss,#Manager,Always,...)
proposition(#Manager,#Manager,Manager,#Manager,Always,...)
proposition(#7,#Manager,*instanceof,#Class,Always,...)
proposition(#8,#Manager,*isa,#Employee,Always,...)
proposition(#Department,#Department,Department,#Department,Always,...)
proposition(#9,#Department,*instanceof,#Class,Always,...)
proposition(#10,#Department,head,#Manager,Always,...)
```

Instantiation to the class `Individual` is implicitly given by the structure of the three individual propositions `#Employee`, `#Manager` and `#Department`. Analogously, the attributes `#3`, `#4`, `#5`, `#6` and `#10` are automatically regarded as instances of the class `Attribute`. The propositions `#1`, `#2`, `#7` and `#9` are instances of the class `InstanceOf`, and `#8` is an instance of the class `IsA`.

The next frame states that the individual object `mary` is an instance of the class `Manager`. `Mary` has four attributes labelled `hername`, `earns`, `advises` and `currentdept` which are instances of the respective attribute classes of `Employee` with labels `name`, `salary` and `dept`.

```
Individual mary in Manager,Token with
      name
            hername: "Mary Smith"
      salary
            earns: 15000
      dept
            advises: PR;
            currentdept: R&D
end mary
```

The corresponding propositions are:

```
proposition(#mary,#mary,mary,#mary,Always,...)
proposition(#Employee1,#mary,*instanceof,#Manager,Always,...)
proposition(#Employee2,#mary,*instanceof,#Token,Always,...)
```

```
proposition(#Employee3,#mary,hername,"Mary Smith",Always,...)
proposition(#Employee4,#Employee3,*instanceof,#3,Always,...)
proposition(#Employee5,#mary,earns,15000,Always,...)
proposition(#Employee6,#Employee5,*instanceof,#4,Always,...)
proposition(#Employee7,#mary,advises,#PR,Always,...)
proposition(#Employee8,#Employee7,*instanceof,#5,Always,...)
proposition(#Employee10,#mary,currentdept,#R&D,Always,...)
proposition(#Employee11,#Employee10,*instanceof,#5,Always,...)
```

We assume two existing instances of Department namely `PR` and `R&D` with corresponding oids `#PR` and `#R&D`:

```
Individual PR in Department,Token
end PR

Individual R&D in Department,Token
end R&D
```

The attribute categories `name`, `salary` and `dept` must be defined in one of the classes of `mary`. In this case `mary` is also instance of `Employee` due to the following axiom which defines the **inheritance** of class membership in Telos, and hence can instantiate these attributes:

Specialization axiom

> ***The destination ("superclass") of a specialization inherits all instances of its source ("subclass").***

An example is the specialization `#8`: all instances of `Manager` (including `mary`) are also instances of `Employee`. Telos enforces **typing** of the attribute values by the following general axiom:

Instantiation axiom

> ***If p is a proposition that is instance of a proposition P then the source of p must be an instance of the source of P, and the destination of p must be an instance of the destination of P.***

For example, `"Mary Smith"` must be an instance of `String`. The individual `mary` also shows another feature : attribute classes specified at the class level do not need to be instantiated at the instance level. This is the case for the `boss` attribute of `Employee`. On the other hand, they may be instantiated more than once as e.g. `dept`.

In some cases for attribute categories occuring in a frame the corresponding objects which are instantiated by the concrete attributes, can not uniquely be determined[1] . This multiple generalization/instantiation problem is solved[2] by the following condition which must hold for Telos knowledge bases:

Multiple generalization/instantiation axiom

> ***If p1 and p2 are attributes of two classes c1 and c2 which have the same label component l, and i is a common instance of c1 and c2 which has an attribute with category l, then there must exist a common specialization c3 of c1 and c2 with an l labelled attribute p3 which specializes p1 and p2, and i is instance of c3. Particularly if c1 is specialization of c2 and p1 is specialization of p2, c1 and p2 already fulfill the conditions for c3 and p3.***

---

[1] Subsection 2..2 contains an example for this problem in the context of linking logical formulas to Telos objects.

[2] For specialization relationships between two objects we need an axiom similar to the instantiation axiom which requires specialization relationships between their sources and destination components. [JEUS92] contains the complete axiomatization.

Telos treats all three kinds of relationships (`attribute`, `isa`, `in`) as objects. Thus each attribute, instantiation or generalization link of `Employee` may have its own attributes and instances. For example, each of the four `Employee` attributes is an instance[3] of an attribute class denoted by the label `attribute` but can also have instances of its own. The attribute with label `earns` of `mary` is an instance of attribute `salary` of class `Employee`. Syntactically, attribute objects are denoted by appending the attribute label with an exclamation mark to the name of some individual. The relationship between `salary` and `earns` could be expressed as

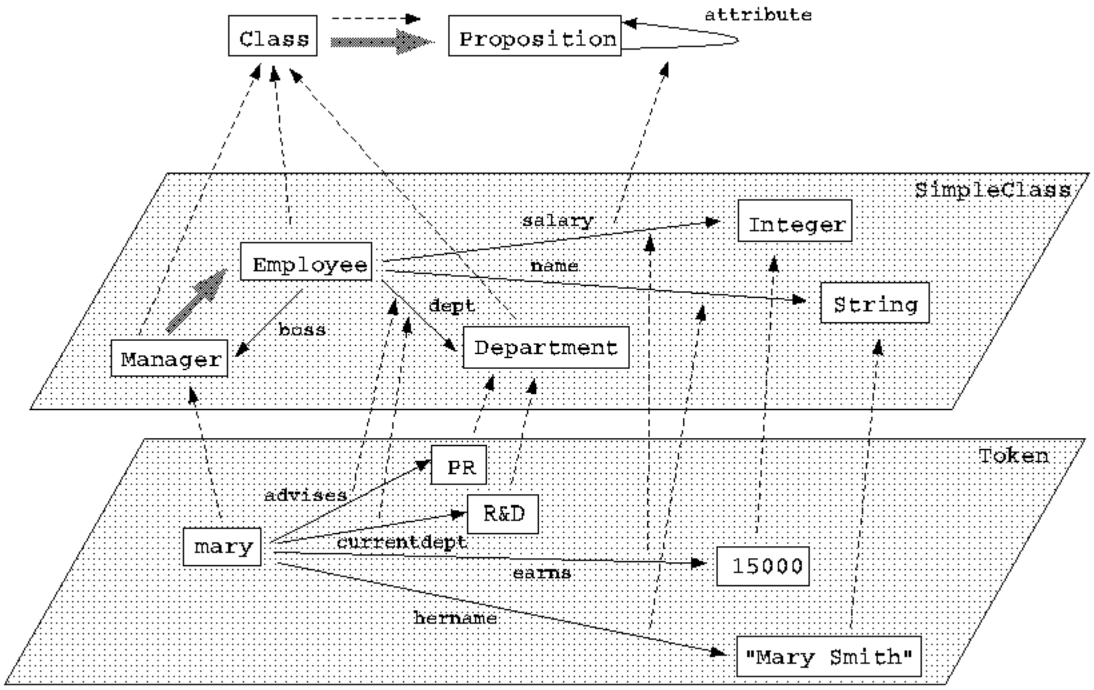
```
Attribute mary!earns in Employee!salary
end mary!earns
```

Instantiation links are denoted by "->" and specialization links by "=>":

```
InstanceOf mary->Manager
end mary->Manager

IsA Manager=>Employee
end Manager=>Employee
```

Fig. 2-1 shows as the second view on propostions the **graphical** semantic network representation of `mary` and her relationships to the other example objects.



2-1: Graph representing the example proposition network

Individual objects are denoted as nodes of the graph, instantiation, specialization, and attribute relationships are represented as dotted, shaded, and labelled directed arcs between their source and destination components.

[3] These instantiations were left out in the set of propositions for the employee example above.

Telos propositions have two temporal components: the **validity time**[4] and the **belief time**. The belief time of a proposition is not assigned by the user but by the system at **transaction time** of an update (`TELL` or `UNTELL`). *ConceptBase* uses right-open and closed predefined time intervals. Example:

```
proposition(#mary,#mary,mary,#mary,
            Always,tt(second(1992,1,11,17,5,42)))
```

The object `mary` is valid at `Always` and it is believed since 17:05:42 on January 11, 1992. Right-open belief times indicate objects that are part of the "current" knowledge base and closed intervals (denoted by binary tt-terms) indicate "historical" objects. See also operations TELL and UNTELL in section 4.

2.2 The Predicative Sublanguage CBL

The *ConceptBase* predicative language CBL [JK90] is used to express integrity constraints, deductive rules and queries. The variables inside the formulas have to be quantified and assigned to a "type" that limits the range of possible instantiations to the set of instances of a class. *ConceptBase* offers a set of literals for the predicative language defined on top of a Telos knowledge base. Usually, this knowledge base is restricted by a rollback time:

KB_{rbt} `= {proposition(oid,x,l,y,t,tt) in KB | rbt during tt}`

The value of the rollback time depends of the kind of formula to be processed: integrity constraints are evaluated on the current KB (`rbt=Now`=the smallest time interval that contains the current time), the rollback time of queries is usually provided together with the query (see method ASK in section 3).

The following three literals[5] are the most important ones of CBL:

1) `(x in c)`
   The object `x` is an instance of class `c`.

2) `(c isA d)`
   The object `c` is a specialization (subclass) of `d`

3) `(x l y)`
   The object `x` has an attribute to object `y` and this relationship is an instance of an attribute category with label `l`. Structural integrity demands that the label `l` belongs to an attribute of a class of `x`.

---

[4] As mentioned in section 1 we plan to remove the validity time from the data model and offer a calculus for this time component as optional extension of the system. So in this manual we don't give detailed hints for the use of validity time in Telos. Nevertheless this feature can still be used as documented in the manuals of earlier releases of ConceptBase but you shouldn't do that if you want to preserve compatibility of your Telos models with future releases.

[5] The former versions of these literals `InstanceOf`, `IsA`, `AttrValue` known from CB3.0 are still available but we recommend not to use them any more. The same holds for the literal `AttrId(x,l,id,t)` which states that object `x` has an attribute `id` which is an instance of an attribute category with label `l` at time `t`

In order to avoid ambiguity, neither `in` and `isa` nor the logical connectives `and` and `or` are allowed as attribute labels[6] .

The next literals are second class citizens in formulas. In contrast to the above literals they cannot be assigned to classes of the Telos knowledge base. Consequently, they may only be used for testing, i.e. in a legal formula their parameters must be bound by one of the literals 1 - 3.

4) `LT(x,y),GT(x,y),LE(x,y),GE(x,y),EQ(x,y),NE(x,y)`
Comparison literals like in FORTRAN. Note that `x` and y must be instances of `Integer` or `Real`. Equivalent infix representations are
`x < y, x > y, x <= y, x >= y, x = y, x <> y`

5) `IDENTICAL(x,y)`
The objects `x` and `y` are the same. The infix representation is
`x == y`

The exact syntax of CBL is given in appendix A2.2. The **types of variables** (i.e. quantified identifiers) are interpreted as instantiations:

```
.  forall x/C F
.  exists x/C F
```

are shorthands for the untyped formulas

```
.  forall x (x in C) ==> F
.  exists x (x in C) and F
```

We demand that each variable is quantified exactly once inside a formula. This is no real restriction: in case of double quantification rename one of the variables. More important is a restriction similar to static type checking in programming languages that demands a strong relationship between formulas and the knowledge base.

Formula Assignment Restriction

Each constant (= arguments that are not variables) in a formula `F` must be the name of an existing object in the Telos knowledge base, or it is a constant of the builitin classes `Integer`, `Real`, or `String`. Moreover, for each attribute predicate `(x l y)` occuring in a formula there must be a unique attribute labelled `l` of one class `c` of `x` in the knowledge base.

A formula violating the first part of the restriction would make a statement about something that is not part of the KB. As an example, consider the following formula:

```
forall x/Emplye not (x boss Mary)
```

With the example KB of section 2.1, we find two errors: There are no objects with names `Emplye` and `Mary`.

There are two possible cases to violate the second part of the restriction. The first case is illustrated by an example:

---

[6] For the example of subsection 2.1 among others the ground literals `(mary in Manager)`, `(Manager isA Employee)` and `(mary earns 15000)` are valid facts describing the contents of the KB. We suggest to choose verbs (e.g. `earns` in our example) for attribute labels to get more natural and readable literals.

```
forall x/Proposition y/Integer (x salary y) ==> x < 10000
```

In this case the classes of `x`, `Proposition` and any of its superclasses, have no attribute labelled `salary`. Therefore, the predicate `(x salary y)` cannot be assigned to an attribute of the KB. Instead, one has to specify

```
forall x/Employee y/Integer (x salary y) ==> x < 10000
```

or

```
forall x/Manager y/Integer (x salary y) ==> x < 10000
```

depending on wether the formula applies to managers or to all employees.

The second case is closely related to multiple generalization/instantiation. Suppose, we have new classes `Guest` and `GuestEmployee`:

```
Individual Guest in Class,SimpleClass with
      attribute
            dept: Shop
end Guest

Individual GuestEmployee in Class,SimpleClass
                         isA Guest,Employee
end GuestEmployee
```

The following formula refers to objects of class `GuestEmployee` and their `dept` attribute. The problem is that **two** different attributes, `Employee!dept` and `Guest!dept`, apply as candidates for the predicate `(x dept PR)`:

```
forall x/GuestEmployee (x dept PR) ==> not (x in Manager)
```

In order to solve this ambiguity, we demand that in such cases a common subclass exists that defines an attribute `dept` which conforms to both definitions, e.g.

```
GuestEmployee with
      attribute
            dept: ShopDepartment
end GuestEmployee

Individual ShopDepartment in Class,SimpleClass
                        isA Shop,Department
end ShopDepartment
```

A "legal integrity constraint" is a CBL formula that fulfills the formula assigment restriction. A "legal deductive rule" is a CBL formula fulfilling the same restriction and having the format:

```
forall x1/c1 ... forall xn/cn R ==> lit(a1,...,am)
```

where

. `lit` is a literal of type 1 - 3
. the variables in `a1`,...,`am` are exactly `x1`,...,`xn`

In Telos, rules and constraints are defined as attributes of classes. Use the category `bdmconstraint` for integrity constraints, and the categories `rule`,`bdmrule` for deductive rules. The text of the formula has to be enclosed by the character '`$`'.

Continuing our running example, the following formula is a deductive rule that defines the `boss` of an `Employee`. Note that the variables `e,m` are `forall`-quantified.

```
Employee with
   rule,bdmrule
     BossRule : $ forall e/Employee m/Manager
                  (exists d/Department
                  (e dept d) and (d head m))
                  ==> (e boss m) $
   bdmconstraint
     SalaryBound : $ forall e/Employee b/Manager x,y/Integer
                     (e boss b) and (e salary x) and (b salary y)
                     ==> x <= y $
end Employee
```

The second formula is an integrity constraint that uses the `boss` attribute defined by the above rule. The constraint demands `salaries` of `Employees` not to excceed the `salary` of their boss. Note that you can define different `salaries` for `Employees`. The constraint is on each individual `salary`, not on the sum! Also note that the arguments of the `<=` literal are bound by the two literals with attribute label `salary`. More examples can be found in the file `EmployeeFormulas.doc` in the directory

`$CB_HOME/CB_Exe/Examples/RULES+CONSTRAINTS`

of the *ConceptBase* distribution tape.

2.3 The Query Language CBQL

In *ConceptBase* queries are defined as special classes whose instances are the answers to the query [STAU90]. This section first defines the structural properties of the query language CBQL and then introduces the predicative component. Queries are instances of a system class `QueryClass` which is defined as follows:

```
Individual QueryClass in Class isA Class with
     attribute
          retrieved_attribute: Proposition;
          computed_attribute: Proposition
     attribute,single
          query_rule: MSFOLquery
end Employee
```

If one specifies super-classes to a query class then all answers must be instances of the super-classes, i.e. the answers are restricted by the ranges of the super-classes. Example: "socially interested" are those managers that are member of a union.

```
QueryClass SI_Manager_0 isA Manager,UnionMember end SI_Manager_0

QueryClass SI_Manager isA Manager,UnionMember with
     retrieved_attribute
          union: Union;
          salary: Money
end SI_Manager
```

Note that the super-classes themselves may be query classes which is the first kind of query recombination. The second frame shows the feature of **retrieved attributes** which is similar to projection in relational algebra. Example: one wants to see the name of the union and the salary of socially interested managers. The attributes must be present in one of the super-classes of the query class. In this example, the `union` attribute is obviously inherited from the class `UnionMember` and `salary` is inherited from `Manager`. CBQL demands that retrieved attributes are necessary: each answer must instantiate them. If an object does not have such an attribute then it will not be part of the solution. As usual with attribute inheritance, one may specialize the attribute value class, e.g.

```
QueryClass Well_off_SI_Manager isA SI_Manager with
     retrieved_attribute
         salary: HighSalary
end Well_off_SI_Manager
```

The new attribute value class `HighSalary` must be a subclass of `Money` so that each solution of the restricted query class is also a solution of the more general one. It should also be noted that `HighSalary` can also be another query class. This is the second way of query recombination.

Retrieved attributes and super-classes already offer a simple way of querying a knowledge base: projection and set intersection. For more expressive queries there is an predicative extension, the so-called **query rule**. We use the same many-sorted predicative language as in section 2.2 for deductive rules and integrity constraints and introduce a useful abbreviation:

> Let `Q` be a query class with a query rule `F` that contains the predefined variable `this`. Then the query rule is an abbreviation for the formula
>
> `forall this/Proposition F ==> Q(this)`
>
> *ConceptBase* automatically derives solutions `(this in Q)` from the answer literal `Q(this)`.

This means that `this` stands for any answer object of `Q`. The following example shows how to use it.

```
QueryClass Well_off_SI_Manager1 isA SI_Manager with
     retrieved_attribute
         union: Union
     query_rule
         well_off_rule: $ exists s/HighSalary
                         (this salary s) $
end Well_Off_SI_Manager1
```

Classes occuring in a query rule may be query classes. This is the third way of query recombination. The next feature introduces so-called **computed attributes**, i.e. attributes that are defined for the query class itself but not for its super-classes. The assignment of values for the solution is defined within the query rule. We introduce a second abbreviation:

> If `ca` is the label of a computed attribute with class `C` of query class `Q` with query rule `F` then `F` stands for the formula
>
> `forall this/Proposition forall v/C`
> `F' ==> Q(this,v).`
>
> The primed `F'` stands for the substitution `F[v/ca]`. If `Q` contains more than one computed attribute then the answer literal `Q(...)` has a corresponding number of additional parameters. *ConceptBase* automatically derives solutions `(this in Q)`, `(this ca v)` from the answer literal.

The following example defines a computed attribute `head_of` that stands for the department a manager is leading.

```
QueryClass Well_off_SI_Manager2 isA SI_Manager with
     retrieved_attribute
         union: Union
     computed_attribute
         head_of: Department
     query_rule
         well_off_rule: $ exists s/HighSalary
                     (this salary s) and
                     (head_of head this) $
end Well_off_SI_Manager2
```

Like retrieved attributes computed attributes are necessary, i.e. any solution of a query with a computed attribute must assign a value for this attribute. Example:

```
Individual Hubert in Well_off_SI_Manager2 with
     union
        : IGM
     head_of
        : Production
end Hubert
```

Recursion can be introduced to queries by using recursive deductive rules or by refering recursively to query classes. The example asks for all direct or indirect bosses of Bill:

```
QueryClass BillsMetaBoss isA Manager with
     query_rule
         billsBosses:
            $ (Bill boss this) or
              exists m/Manager
                (m in BillsMetaBoss) and
                (m boss this)$
end BillsMetaBoss
```

Further examples are on the distribution tape in the directory `$CB_HOME/CB_Exe/Examples/QUERIES`.
Queries are represented as Telos classes and consequently they can be stored in the knowledge base for future use. It is a common case that one knows at design time **generic queries** that are executed at run-time with certain parameters. CBQL supports such parameterized queries:

```
Individual GenericQueryClass isA QueryClass with
     attribute
        parameter: Proposition
end GenericQueryClass
```

Generic queries are queries of their own right: they can be evaluated. Their speciality is that one can easily derive specialized forms of them by substituting the parameters. An important property is that each solution of a spezialized form is also a solution of the generic query. This is a replay of the inheritance paradigm. The example shows that parameters can be retrieved or computed attributes at the same time:

```
GenericQueryClass What_SI_Manager isA Manager,UnionMember with
     retrieved_attribute,parameter
        salary: HighSalary;
        u: Union
     computed_attribute,parameter
        head_of: Department
     query_rule
        well_off_rule: $(this union u) and (head_of head this)$
end What_SI_Manager
```

There are two kinds of specializing generic query classes:

1) Specialization of a parameter "`[a:C']`"
   Example: `What_SI_Manager[salary:TopSalary]`
   In this case `TopSalary` must be a subclass of `HighSalary`. The solutions are those managers in `What_SI_Manager` that not only have a high but a top salary.
2) Instantiation of a parameter "`[v/a]`"
   Example: `What_SI_Manager[Research/head_of]`
   The variable `head_of` is the replaced by the constant `Research` (which must be an instance of `Department`).

One may also combine several specializations, e.g.
`What_SI_Manager[salary:TopSalary,Research/head_of].`
The specialized queries can occur in other queries in any place where ordinary classes can occur, e.g.

```
QueryClass FavoriteDepartment isA Department with
     retrieved_attribute
         head: What_SI_Manager[10000/salary]
end FavoriteDepartment
```

Parameters that don't occur as `computed` or `retrieved` attributes are interpreted as existential quantifications if they are not instantiated.

The current implementation provides two **executable representations** for query classes: SLDNF (Prolog) and MS (Magic set). The first one is efficient in time and space but currently not able to handle recursion. There is a fact `QueryRep(.)` in the file `CB_Exe/GlobalParameters.pro` of the *ConceptBase* distribution tape that specifies the default representation. If it is set to SLDNF one may use the `Magic` class to override it:

```
QueryClass Q in Magic ...
```

Similar deductive rules can also be represented in the magic format. This is done by using the `mrule` instead of the `rule` category:

```
...
mrule,bdmrule
   myrule: $ forall ... $
   ...
```

Both representations can be mixed but one should notice that query classes and rules in SLDNF format don't have access to the MS format of other queries and rules.

abcd

3 ConceptBase X11 Usage Environment

The *ConceptBase* X11 Usage Environment provides tools to manage conceptual models under the X window system, version 11 release 5. The X Window System was chosen to make *ConceptBase* clients available to the broad range of machines which support this protocol. Our implementation has so far concentrated on basic functionalities and tools which are subsumed under the *General ConceptBase Usage Environment.*

This chapter is divided into three sections. The first one deals with the *ConceptBase Tool Bar* which provides some management facilities, the second one with the so called *Basic Tools* (*Telos Editor*, *Display Instances*, *Display Queries*, *TextBrowser* and *GraphBrowser*) and the third one contains an example dialogue with the system.

3.1 ConceptBase Tool Bar

The *ConceptBase Tool Bar* (figure 3-1) is always present on the screen during a session. The *Tool Bar* consists of five sections (**system**, **models**, **editing**, **displaying**, and **browsing**). At the beginning of a session the panel of the system section offers two choices:

Connect CB Server - establishes a connection to a *ConceptBase* server
Stop Usage Environment - terminates the current session

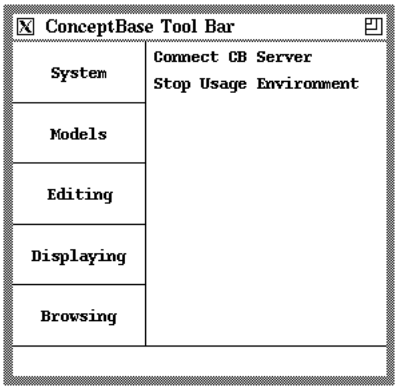


Fig. 3-1: The *ConceptBase Tool Bar* showing the panel of the system section before connecting to a *ConceptBase* server.

In order to connect the usage environment to a *ConceptBase* server it is necessary to specify the name of the host machine running the *ConceptBase* server to be connected and the port number of the server. Note, that port numbers have to be between 2000 and 9999. For convenience we provide two environment variables - `CB_SERVER` and `CB_PORTNR` - defining the defaults presented at the first time **Connect CB Server** was chosen. The values specified within the interaction window appearing were recorded and presented as default at the next time **Connect CB Server** is chosen. In addition to these variables one might directly specify port number and host machine as command line parameters of the *ConceptBase Tool Bar*.

After connecting to a *ConceptBase* server each section of the *ToolBar* provides different buttons from which all the other facilities and tools can be invoked. These menus are :

- **System** panel which deals with client server administration

  Disconnect CB Server - terminates the connection to a *ConceptBase* server;
  Connect CB Server - terminates the current connection and establishes a new connection to a *ConceptBase* server;
  Stop Usage Environment - terminates the current session;
  Stop CB Server - stops the *ConceptBase* server your usage environment is currently connected to[1] ;
  Report Clients - displays all clients using the *ConceptBase* server your usage environment is currently connected to;
  Kill all ErrorQueues - clears all connections between clients and error windows or answer windows[2] ;

- **Model** panel which offers facilities to load Telos objects from external files. Again, environment variables are provided for convenience. `CB_APPDIR` specifies the directory applications are loaded from; `CB_APP` specifies the application to be loaded. `CB_MODELDIR` specifies the directory conceptual models[3] are loaded from, `CB_MODELS` specifies a comma-separated list of models (= filenames without extension) to be loaded into the KB.

  Add Models - adding Telos objects (in frame representation) from external files to the current application;
  Load Applications - loading an existing application from external files or setting up a new application;

- **Editing** panel which offers tools for editing objects

  Telos Editor - edits the syntax of Telos objects, described in section 3.2.1;
  CoAuthor - supports the group process of authoring hypermedia documents, described in appendix 4;
  Hypertext Editor - edits hypermedia documents, described in section 3.2;

- **Display** panel which offers facilities for displaying instances of an object (class)

  Display Instances - displays instances of an class, described in section 3.2.2;
  Display Queries - displays instances of `QueryClass`, described in section 3.2.3;

- **Browsing** panel which offers facilities for browsing through the knowledge base

  Browse textually - invokes the *TextBrowser,* described in section 3.2.4;
  Browse graphically -invokes the *GraphBrowser* described in section 3.2.5;
  Query - offers an interface to *ConceptBase*‘ query language;

---

[1] This will terminate the ConceptBase server even if other users are still using it. One can check for other users by using the Report Clients facility.

[2] Due to incorrect termination of error windows, answer windows, or client tools some of the communication channels might be blocked. To free the channels invoke this menu item.

[3] As mentioned in section 1 we plan to introduce a module concept in our system which is suited to manage models as subunits of applications. In this version of ConceptBase our notion of models is simply a collection of Telos objects in frame representation stored in files with a `sml` extension.

3.2 Basic Tools

3.2.1 Telos Editor

The *Telos Editor* is used to display, modify and create Telos objects. After calling the editor a window appears, querying for the identifier of the object to be edited and for the rollback time (section 2.2). The Telos syntax of the specified object is then shown.

The editing facilites are based on the text package provided by the Andrew Toolkit (including *cut*, *copy* and *paste*). A panel offers additional buttons:

tell - the current content of the editor is communicated to the *ConceptBase* server, where it is ‘integrated’ into the knowledge base;
untell - the current content of the editor is communicated to the *ConceptBase* server, where it is ‘removed’ from the current knowledge base and ‘entered’ into the historical knowledge base;
ask - the current content of the editor is communicated to the *ConceptBase* server, where it is interpreted as a query, an answer is computed and passed to a specific answer window;
new Object - is used to edit another object. An interaction window queries for the object identifier of the new object and the rollback time. If the content of the editor window was changed since the last **tell** or **untell** operation, a pull-up window appears to inform the user and waits for a confirmation (**Cancel** or **Nerver Mind**);
help - a window appears containing a description of the editor facilities;
quit - terminates the *Telos Editor*;

During the **tell** and **untell** operations scanning, parsing and transforming the content of the editor to SML-fragments (syntax trees for Telos frames) is performed. Syntactical and semantical correctness (e.g. satisfaction of instantiation axioms) is also checked by the object processor during the **tell** and **untell** operations. Errors are reported via an error window[4]

3.2.2 Display Instances

The **Display Instance** facility is used to display all instances of a given object direct ones as well as inhereted ones. This facility is a specialization of the textual browsing tool (*TextBrowser* described in section 3.2.4). After invoking **Display Instance** only an object name and the rollback time is requested. No links and directions have to be specified.

Each displayed object can be selected by pointing to it and clicking the left mouse button. After selecting an object the following buttons are available:

TelosEditor - calls the *TelosEditor* with the selected object (section 3.2.1);
DisplayInstances - shows the instances of the selected object;

---

[4] The communication between the *Telos Editor* and the *Error Window* or *Answer Display* may break due to incorrect termination of either partner. If this occurs, errors are not transfered to the *Error Window* and the message "Unable to pass error message to *Error Window*" appears in the *Editor*’s message line. Choose **Kill all Error Queues** from the system panel to clear all communication lines.

GraphicalBrowser - calls the *GraphBrowser* with the selected object as root (section 3.2.5);
CoAUTHOR - calls the *CoAUTHOR tool* (see appendix 4);
Get Document - calls the hypertext editor with the selected object (see appendix 4);
Help - a window appears containing a description of the selection window's facilities;
quit - terminates **Display Instance**.

3.2.3 Displaying Queries

Display Queries is used to display all instances of the system class `QueryClass` (and its specializations). All query objects can be selected. The tool provides a panel with the following buttons:

TelosEditor - calls the *TelosEditor* with the selected object (section 3.2.1);
Ask - gets the answer for the selected query from the *ConceptBase* server. The answer is displayed within a specific *Ask Replies* tool;
Help - a window appears containing a description of the selection window's facilities;
quit - terminates **Display Queries**.

3.2.4 TextBrowser

The *TextBrowser* queries the user for a specification of the object to be displayed. This specification covers one hierarchical level and consists of the root object (origin), the orientation of the links, (i.e. following links `coming_out_of` or `going_into` the source object) and the label of those links (e.g. '`*instanceof`', '`*isA`'). The links matching this specification are then listed in a table of the form:

```
SourceObject            Label        DestinationObject
```

Each part of the displayed links can be selected. After selecting a source, label or destination the possibility of calling a new *TextBrowser* or *GraphBrowser* with the selected object as root object is offered (see the example dialogue in section 3.3).

3.2.5 The GraphBrowser

The *GraphBrowser* requires only an object identifier and then draws the corresponding object as an initial node of a directed acyclic graph (DAG). By clicking the left mouse button every displayed object can be selected. A selected object can be moved by dragging the object with the right mouse button. Note, that selecting and moving objects is available for nodes and links. The following buttons are provided:

erase node - deletes the selected object (node or link) from the display.[5]
any - displays all objects connected to the selected object by a link corresponding to a user-defined specification of orientation and label (see section 3.3 for an example).

[5] Erasing an object has no impact on the content of the knowledge base.

show attributes - displays all the direct attributes (not the inherited ones);
show instances - displays all the direct instances (not the derivable ones); not available if the selected object is a link;
show classes - displays all classes the selected object is an instance of; not available if the selected object is a link;
show subclasses - displays one level of subclasses (i.e. subclasses of a subclass of the selected node are not displayed); not available if the selected object is a link;
show superclasses - displays one level of superclasses (i.e. superclasses of a superclass of the selected node are not displayed); not available if the selected object is a link;
Help - a window appears containing a description of the *GraphBrowser*'s facilities;
quit - terminates the *GraphBrowser*.

Additional buttons are provided for invoking the *TelosEditor* (section 3.2.1), displaying instances (section 3.2.2) or invoking another *GraphBrowser* with the selected object. If no object is selected, there is an additional option **Add Node**. Chosing it a new object with no connection to the existing graph is introduced at the position of the last left mouse click.

The way how nodes or links are displayed in the *GraphBrower*'s window is modeled within the knowledge base by specializations of the Telos classes `NODE` and `EDGE` which are instances of the class `AnswerRepresentation`. In particular, the format of the standard *GraphBrowser* is defined by the Telos classes `X11GraphBrowserNODE` and `X11GraphBrowserEDGE`, which are predefined system classes.

```
Class X11GraphBrowserEDGE in AnswerRepresentation isA EDGE with
  graphicalType
    gT1  : MetametaClass;
    gT2  : MetaClass;
    gT3  : SimpleClass;
    gT4  : Token
end X11GraphBrowserEDGE
```

This format defines four graphical types: `MetametaClass`, `MetaClass`, `SimpleClass`, and `Token`. For a given object to be displayed in the *GraphBrowser*'s window its instantiation relation to one of these classes determines its graphical type. If the object at hand is not an instance of any of the four classes, `Proposition` is returned as the graphical type. For example, the object `Employee` introduced in 2.1 gets the graphical type `SimpleClass`. In order to define the graphical type of an object unequivocal, each attribute link of `X11GraphBrowserEDGE` of the attribute catagory `graphicalType` must get an *order value* indicating the ranking of the corresponding graphical type. The attributes must be numbered consecutively starting with number 1 (= highest order value). The order value determines to which graphical type an object belongs if it is an instance of more than one graphical type: to the one which corresponding attribute defines the highest order value (= lowest number). For example, the attribute `gT1` got the order value 1, i.e. every instance of the object `MetametaClass` gets the graphical type `MetametaClass`, though it might be an instance of other graphical types.

```
X11GraphBrowserEDGE!gT1 with
  orderValue
    v:1
end
```

In the same manner as `X11GraphBrowserEDGE`, there is a Telos class `X11GraphBrowserNODE` which has to be defined in the same way as `X11GraphBrowserEDGE`. The reason for this is, that there are queries asking for links to be displayed, eg. the **show attributes** option described above. Other queries ask for a single object to be displayed, eg. the **Add Node** option. In order to get an answer format which can be easily processed by the *GraphBrowser*, the two formats have been introduced. To create user-defined formats the following steps must be performed:

> 1) Define one new class for the the graphical types of edges and one for the graphical types of nodes by telling new specializations of the classes `EDGE` and `NODE`.
> 2) Assign a unique order value to every attribute of the objects defined in 1) starting with number 1. Note that the values have to be continuos for edges and for nodes.

An example on how to create user-defined formats can be found in appendix A3.2.

The mapping of graphical types to concrete graphical objects displayed on the screen is done within the *GraphBrowser*'s initialization file, named `.graphbrowserinit`[6] , which is splitted into two parts. The first part contains descriptions of the buttons available within the *GraphBrowser*'s panel. This part should not be changed in order not to loose any options. The second part defines the mapping from graphical types defined within the knowledge base to concrete patterns and shapes displayed on the screen[7] . A line of the second part of the initialization file consists of four parts:

```
GraphicalType, Pattern,  Shape, MenuMask
```

with the following meanings of the fields:

`GraphicalType`:	the name of a Telos class which is an attribute of `X11GraphBrowserEDGE` and `X11GraphBrowserNODE`;	
`Pattern`:	one of the following: `none, black, gray, dashed_link, double_link`;	
`Shape`:	one of the following: `rectangle, oval, circle, icon, link`;	
`MenuMask`:	a list of mask specifications separated by '|' indicating a bit wise OR; note, that the mask specified for a given graphical type determines what options are available when selecting an object of that graphical type; possible masks are:	
	`GB_NODE`	options **Show Instances**, **Show Classes**, **Show Subclasses**, **Show Superclasses** and **Erase Node** are available for this graphical type;
	`GB_LINK`	option **Erase Link** is available for this graphical type;

[6] The program first tries to find this file in the actual directory. If there exists no such file the default one is used.

[7] Note that the mapping from graphical types to concrete patterns must be unique within one initialization file.

`GB_SOMETHING`	options **Show Attributes** and **Any** are available for this graphical type;
`GB_UNSELECT`	option **Add Node** is available for this graphical type;
`GB_TOOLS`	options **Telos Editor**, **Graph Browser**, and **Display Instances** are available for this graphical type.

The optional environment variable `CB_ASK_GBFORMAT` is used to enable the change of the way how links and nodes are displayed in the *GraphBrowser*'s window within one *ConceptBase* session. If the variable is set to `ON`, an interaction window will appear before the *GraphBrowser* starts up. The interaction window asks for the *node format* and the *edge format*. The *node format* is the name of the object defining the graphical types of the nodes (e.g. `X11GraphBrowserNODE`), the *edge format* is the name of the object defining the one for the edges (e.g. `X11GraphBrowserEDGE`). For the new GraphBrowser these objects determine the graphical types. Note that these objects must exist in the knowledge base and that a mapping of the graphical types to concrete patterns must be defined in the initialization file. If the variable is not defined or it is set to `OFF` the format of the standard *GraphBrowser* is used.

3.3 Example Dialogue with ConceptBase

In this section we demonstrate how to deal with the usage environment. To illustrate the steps to be done we use an example model. It consists of a few classes including `Employee`, `Department`, `Salesman`. The class `Employee` has the attributes *name*, *salary*, *dept,* and *boss*. In order to create an instance of `Employee` one may specify the attributes `salary`, `name`, and `dept`. The attribute `boss` will be computed by the system using the `bossrule`. There is also a constraint which must be satisfied by all instances of the class `Employee which` specifies that no employee may earn more money than its boss. The Telos notation for this model is given in appendix 3.

3.3.1 Starting ConceptBase

To start *ConceptBase* we use two terminal windows, one for the *ConceptBase* server and one for the usage environment. We start the *ConceptBase* server by typing the command ´**goCBserver 4200**' in a terminal window of, lets say machine **alpha**. Then we start the usage environment with the command '**toolbar**' in the other window.

3.3.2 Connecting the Usage Environment to the ConceptBase Server

Next we establish a connection between the *ConceptBase* server and the usage environment. This is done by choosing the opition **Connect CB Server** from the **System** panel of the *ConceptBase Tool Bar.* An interaction window appears querying for the host name and the port number of the server (i.e. the number we have specified within the command ´**goCBserver 4200**'). To connect the *ConceptBase Tool Bar* directly to the *ConceptBase* server start the *ConceptBase Tool Bar* calling **toolbar 4200 alpha**. Please refer to appendix 1 for further information.

3.3.3 Loading Objects from external Files

A collection of external files, called application, is used to store objects in a persistent way. To load an existing application call the menu item **Load Application** from the **Models** panel. Then you have to specify the name of the application and the absolute path name of the directory where this application is located. Then the following will be performed:

- The current application is closed, i.e. it is removed from the knowledge base. After booting the *ConceptBase* server the current application is set to a default file residing in the `/tmp` directory of the server host.
- If there exists already an application with the specified name in the specified directory (consisting of at least one file with extension `prop` and optionally of files with extensions `rule, hprop, builtin`) this application will be loaded into the knowledge base. Otherwise you will start with an empty application. An error message will occur if the specified directory doesn't exist.
- In the first case backup files with extension `CBback` of the application files will be created so that you can undo all updates of your last *ConceptBase* session by simply replacing the application files by the corresponding `CBback` files.

An empty application is set up by calling **Load Application** specifying a new name for the application and a directory where it should be located. Note: It is recommended to use separate directories for each application. (The next version of ConceptBase will support this by a module concept.)

If the fact `UpdateMode/1` from module `GlobalParameters.pro` of the CB server is set to `persistent` - its default value - (commandline parameter `-u` for the runtime version; compare appendix 1) the application can interactively be modified using the editor commands TELL/UNTELL. All activities like creating new objects or modifying existing objects are stored simultanously to the PROLOG KB **and** to the specified application files so that persistence of interactive updates is guaranteed. This is the usual way of extending an application. If `UpdateMode` is set to `nonpersistent` all interactive updates are lost when the application is changed.

Another way of extending applications is to load Telos objects (expressed in frame syntax) stored in Unix files with the extension `sml`. Call the menu item **Add Models** from the **Models** panel to include these objects into the *ConceptBase* knowledge base. In our example the files `Employee.prop` and `Employee.rule` build the application containing the Employee example. They have been created in the below described way for setting up applications. The files of the Employee example can be found in the directory

```
$CB_HOME/CB_Exe/Examples/QUERIES
```

where `$CB_HOME` is the *ConceptBase* installation directory. Note, that you only have to type the name of the files without their extension. The following files contain the objects of the Employee example expressed in frame syntax: `Employee_Classes.sml`, `Employee_Instances.sml`, `Employee_Queries.sml`. If there were not an existing application `Employee` we could create a new empty application and then add the objects in these files by using **Add Models**. During this load operation *ConceptBase* would check for syntactical and semantical correctness and report all errors to an error window as is done when updating the KB interactively using the editor. This error window collects errors reported since the beginning of the session. By pressing the delete button the user can delete all error messages recorded so far.

3.3.4 Displaying Objects

To display all instances of an object, e.g. the class `Employee`, we invoke the display instance facility by selecting the item **Display Instances** from the **Displaying** panel of the *ToolBar*. In the interaction window we specify `Employee` as object name and `now` as the rollback time. The instances of the class `Employee` are then displayed (figure 3-2).

After selecting a displayed instance we can either start the *GraphBrowser* with the selected object as root, or the *Telos Editor* for editing the syntax of the selected object. If the selected object is an instance of the class `Idea` we have the additional choice of invoking a design specific tool, the *CoAUTHOR* tool (described in appendix 4).

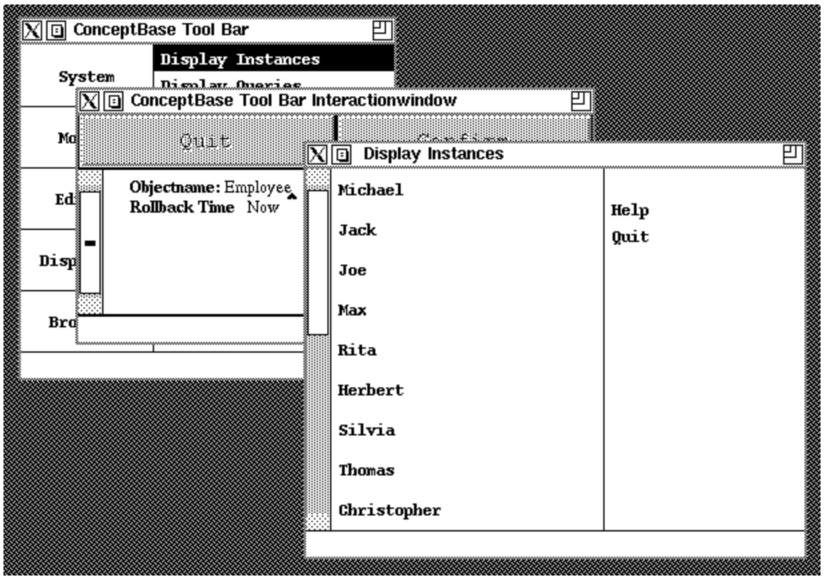


Fig. 3-2: Display of the instances of the class `Employee`

3.3.5 Browsing Objects

For browsing through a loaded application in the KB we use the textual browsing tool or the graphical one. Invoking the *TextBrowser* a focus must be specified. In additional, the orientation (`coming_out_of` or `going_into`) and the label (e.g. '`*instanceof`') must by specified. An example with the system class `SimpleClass` as root is shown in figure 3-3.

The result of the specification is also shown in figure 3-3. We can select any of the displayed objects (nodes or links) in the *TextBrowser* window by clicking the left mouse button over the desired object. After selecting an object a panel is available which offers the choice to call another *TextBrowser* or *GraphBrowser* with the selected object as focus. We select the object `Employee` and choose the item *GraphBrowser* from the provided panel.

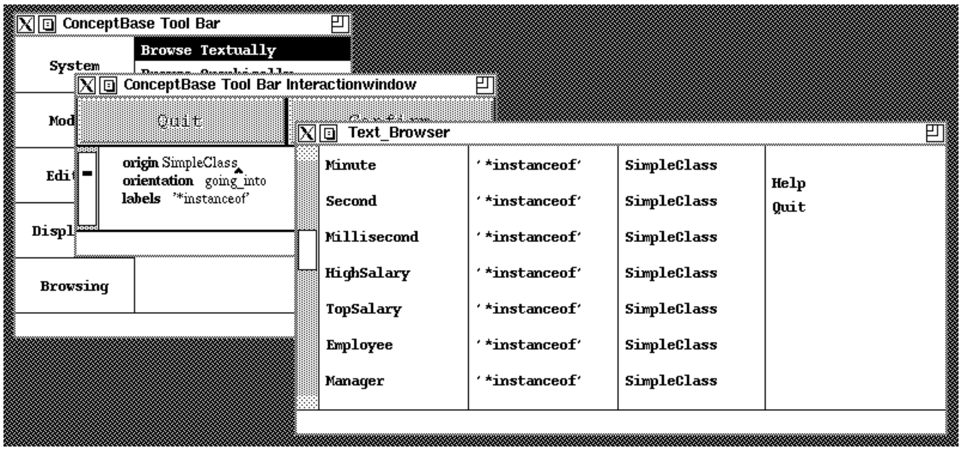


Fig. 3-3: Invoking the *TextBrowser* with the system class `SimpleClass` as focus, the label `'*instanceof'` and the orientation `going_into`.

The *GraphBrowser* (described in section 3.2.5) allows you to expand every displayed object (node or link). We expand the object `Employee` by selecting the node `Employee` and choosing the item **Show Subclasses** from the provided panel. The displayed graph is now expanded (figure 3-4). Notice, that the selected object, the node `Employee`, is marked with small rectangles at each corner.

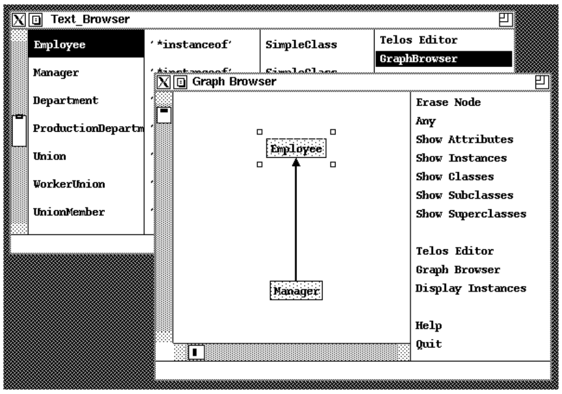


Fig. 3-4: The resulting graph after expanding the node `Employee` with subclasses.

Now we expand the node `Manager`, a subclass of `Employee`, by choosing the menu item **Show Instances**. The resulting graph is shown in figure 3-5.

Now the selected object is the node `Manager`. The instances of `Manager` are shown as white ovals, because they are also instances of the system class `Token`. The nodes `Manager`, `Salesman`, `Employee` etc. are shown as rectangles, since these nodes are instances of the system class `SimpleClass` (see section 3.3 for a full description of the semantics of the graphical objects).

One can move nodes and links by selecting a node or a link and then holding down the right mouse button while moving the cursor to a different position. When the button is released the selected object will be located at the current position and the related links are redisplayed.

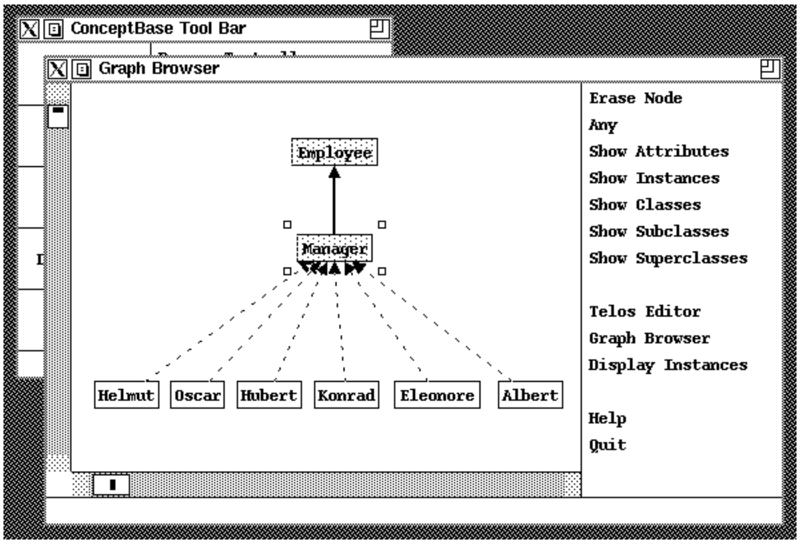


Fig. 3-5: The resulting graph after expanding with the instances of the class `Manager` .

We select the node `Employee` again and choose the menu item **Any**. An interaction window appears querying for the orientation (`coming_out_of`, `going_into`, or simply `c` and `g`) and the label to be considered for the expansion of the displayed graph. We specify `c` as orientation and `'*instanceof'` as label, to see all the classes, where `Employee` is an instance of. The resulting graph gives an overview of the position of the class `Employee` in our knowledge base. To see the attributes of the class `Employee` we choose the menu option **Show Attributes** (the node `Employee` should still be selected). Figure 3-6 shows the resulting graph. Some of the nodes have been manually moved to present a clearer picture.

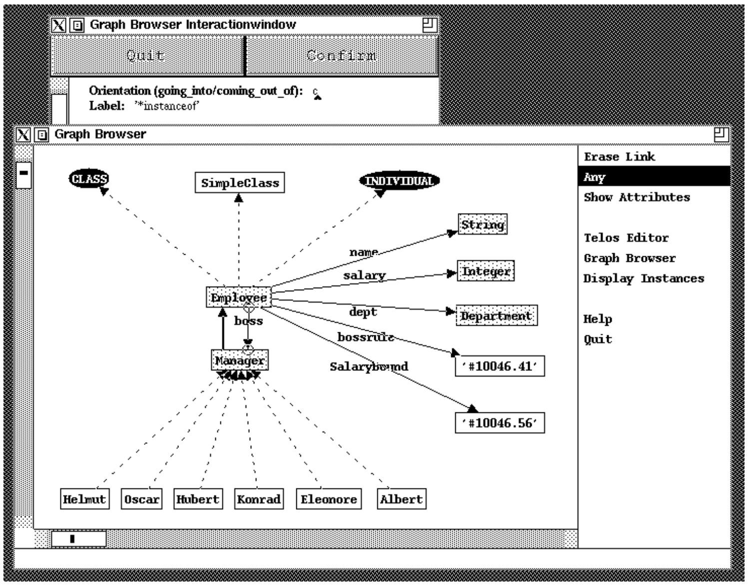


Fig. 3-6: The graph after expanding it with the superclasses and the attributes of the class `Employee`.

3.3.6 Editing Telos Objects

For editing a Telos object we select the desired node, e.g. `Employee`, and call the *TelosEditor* by choosing the **Telos Editor** item from the panel. The input is transfered to the *ConceptBase* server which extracts the Telos frame for the specified object from the knowledge base. The result is transfered to the usage environment and displayed in the TelosEditor shown in figure 3-7.

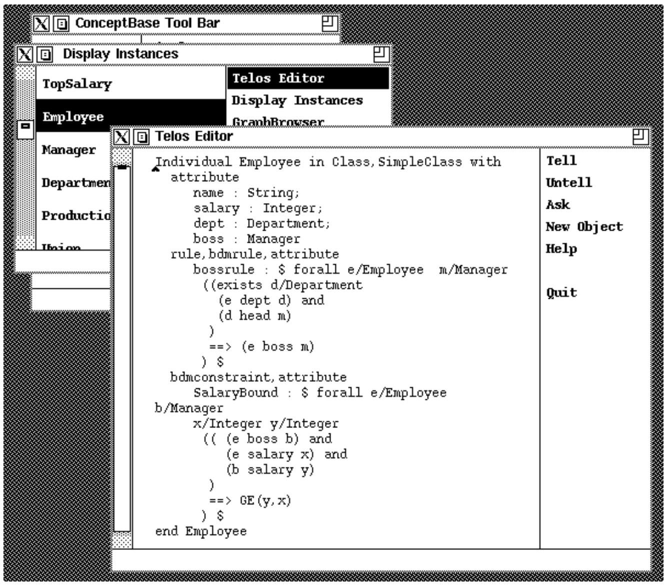


Fig. 3-7: The *TelosEditor* with the object `Employee`.

Now we add an additional attribute, e.g. `education`, to the class `Employee` (for the description of the Telos syntax see appendix 2). The added statement is highlighted in figure 3-8. To show how errors are reported to the user and how to correct them, we have made mistakes in the syntax notation of the added attribute.

By clicking the left mouse button over the **Tell** button the content of the editor is told to the *ConceptBase* server. Syntactical and semantical correctness is checked and the detected errors are reported to an error window. The report resulting from our mistakes by specifiing the syntax of the new attribute is also shown in figure 3-8.

We correct the error by changing the added line from 'education strung' to 'education : String;' and choose the **Tell** menu item again. This time, since there are no further mistakes, the additional attribute is added to the class `Employee`.

We quit the *TelosEditor* (clicking the **quit**) and choose again the item **Show Attributes** from the panel of the *GraphBrowser* window[8] . The additional attribute `education`, added by using the *TelosEditor*, is displayed.

[8] The node Employee should still be selected.

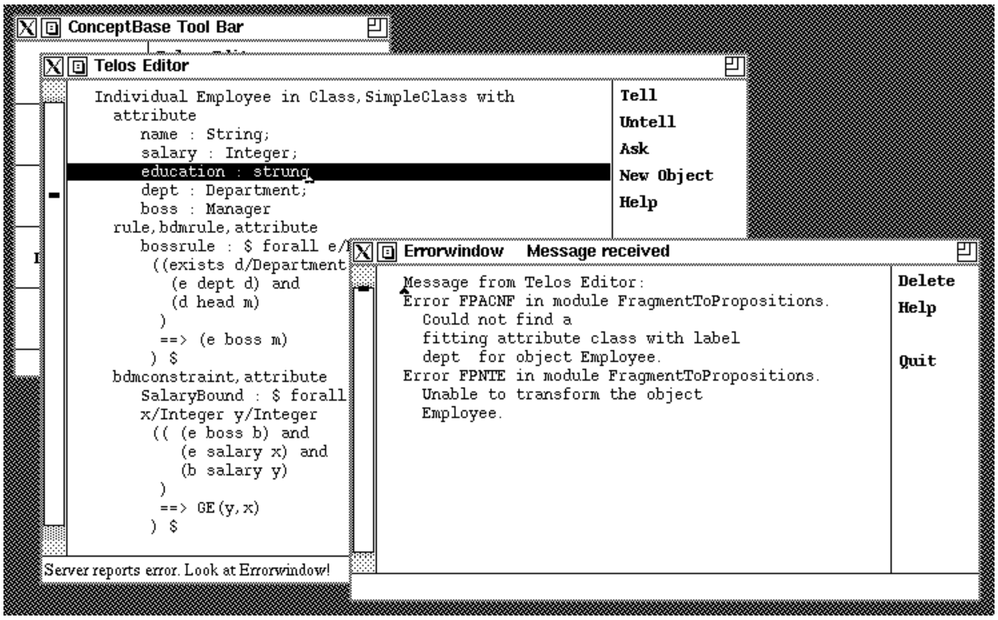


Fig. 3-8: Trying to add an attribute to the class `Employee` with the resulting error report.

3.3.7 Using the Query Facility

We now want to ask the server for all `Employee`s working for `Hubert`. We first clear the window of the *TelosEditor* selecting the whole text and choosing the **Cut** menu item. After tying the new query object (`HubertsEmployees`) we choose the **Ask** menu item. An interaction apperars querying for the format the answer should given and the rollback time. We leave the default values `FRAME` and `Now` indicating that the answer should be displayed as Telos frame and the current state of the knowledge base should be used. The query object is then communicated to the *ConceptBase* Server and the answer is displayed in the *AskReplies* tool. Figure 3-9 shows the *TelosEditor* containing the query object and the *AskReplies* tool with the answer in it.

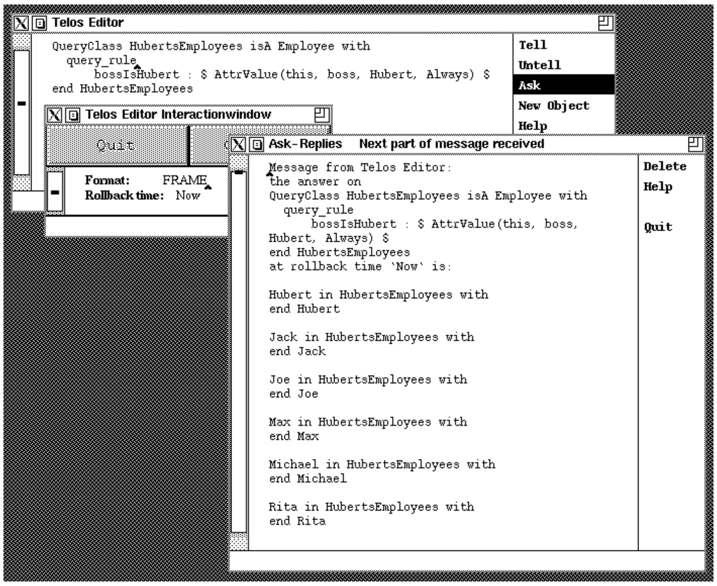


Fig. 3-9: Query object and its answer.

abcd

4 Server Interface

This chapter provides information necessary for communication with the *ConceptBase* server, hereafter called the CBserver. It is possible that the CBserver and the clients 'live' on different machines because communication with the CBserver is realized through a message protocol using inter-process communication (IPC). The following chapter describes this protocol as it is necessary to know for a specialized client which wants to request services from the CBserver. (Readers who intend only to use one of the distributed clients, *ConceptBase* UsageEnvironment for example, may skip this chapter.) You can obtain further informations from the Concept Base Programmers Manual if you want to develop your own specialized client program in PROLOG or C. The Programmers Manual describes PROLOG and C functions which can be used as a library hiding the low-level parts of the communication via IPC.

This chapter is organized as follows: Section 4.1 describes the message protocol which is used to communicate with other processes. Section 4.2 describes the interface to the CBserver, i.e. the data structures and operations which the *ConceptBase* kernel offers and the message protocol which makes these operations accessible to other processes.

A PROLOG demo client `ExampleIpcClient.pro` and example data are distributed with the *ConceptBase* distribution files as a guideline for the communication with a CBserver. This data can be found in the subdirectories

`$CB_HOME/CB_Exe/Examples/CLIENT-SERVER` and
`$CB_HOME/CB_Exe/Examples/QUERIES`

where $CB_HOME refers to the *ConceptBase* installation directory. The `ExampleIpcClient` is a PROLOG program written to illustrate the programmer interface of *ConceptBase*. It corresponds to the box 'Client program' in Fig. 4.1.

All examples in this chapter are taken from a session where the `ExampleIpcClient` communicates with the CBserver.

From a client's point of view the CBserver can be seen as an abstract data type exporting several parameterized operations. These operations comprise methods for storing/retrieving information into/from the KB, methods for establishing and closing the connection to a CBserver and methods for testing the KB. The most important of these methods work on Telos frames. Since the client and CBserver are two different processes a client cannot directly call these methods like procedures but must access them using a message protocol.

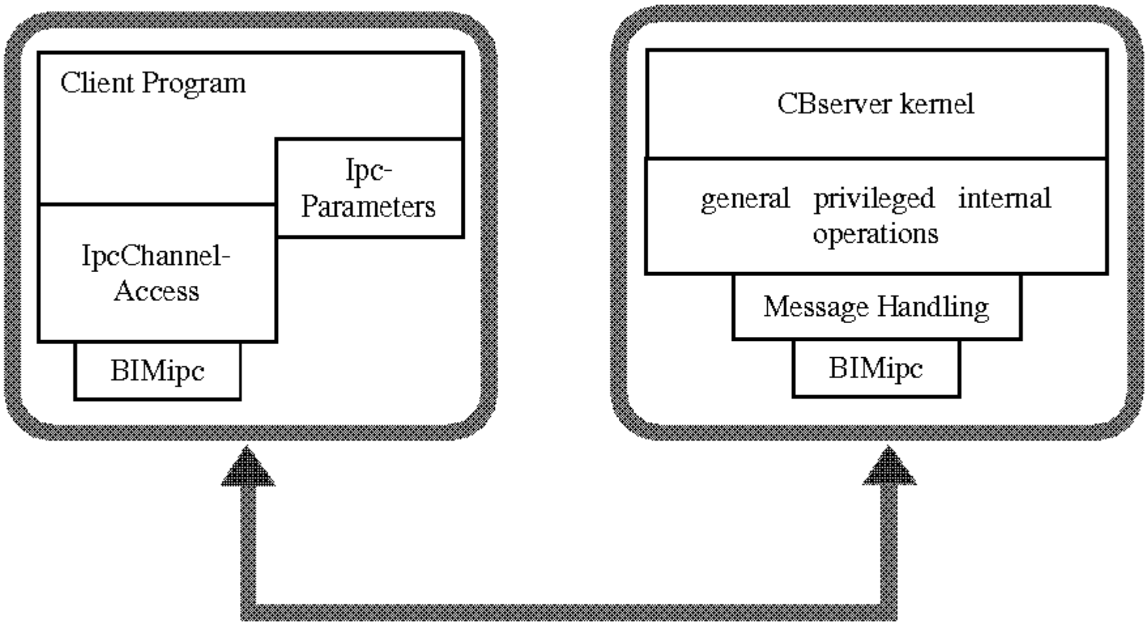


Fig. 4.1: Client-Server Architecture of *ConceptBase*

4.1 Message Format

As already mentioned, any client that wants to use the methods of a CBserver has to communicate with this CBserver according to a message protocol. So called **ipcmessages** can be sent via IPC to the port reserved for this CBserver. The CBserver handles such a message and reports back an answer: the **ipcanswer**. Messages and answers have the following format :

`ipcmessage ( sender, receiver, method, args ) where`

- `sender`: is the identifier for the sender of the message,

- `receiver`: is the identifier for the receiver of the message (usually the CBserver itself, but could be any other client connected to CBserver as well),

- `method`: is one of the methods exported by the CBserver (or a method known to another client which is adressed by the message),

- `args`: are the arguments for method.

`ipcanswer ( sender, completion, return ) where`

- `sender`: is the identifier of the answering program (usually the CBserver since other programs cannot answer directly but only receive the message and send back another message via the CBserver),

- `completion`: signals success (=`ok`) or failure (=`error`) or unability (=`not_handled`) of handling the message

- `return`: contains the return value(s) of the handled message.

Additionally the CBserver administrates message queues for all connected clients. Whenever a client X sends a message to another client Y (which is not CBserver) the CBserver stores this message into the message queue of the client Y and gives back to X

```
ipcanswer ( CBserverID, ok, 'message queued' ).
```

Client Y fetches the message (see method NEXT_MESSAGE below) via a call to CBserver. Then this message can be handled by client Y and the answer will be sent as a new message again via a call to the CBserver to the client X (i.e. it is stored in the message queue of client X by the CBserver).
Remark: If clients are likely to exchange messages they should periodically poll their message queue.

4.2 Methods Exported by the CBserver

The CBserver offers the following methods:

a) general methods:	TELL, TELL_OBJPROC, UNTELL, ASK, HYPO_ASK, LOAD_APPLICATION, NEXT_MESSAGE
b) privileged methods:	STOP_SERVER, REPORT_CLIENTS, REPORT_TOOLCLASS
c) internal methods:	ENROLL_ME, CANCEL_ME, PROLOG_CALL

In the following description of the methods `return` refers to the respective parameter of `ipcanswer(sender, completion, return)`.
In the examples taken from a session where a `ExampleIpcClient` interacts with a CBserver.

```
Method: @
Arguments ([a1,..,an]): @

Responding tool:
Completion:
Return value:
```

denote the prompts of the `ExampleIpcClient`. The first two prompts ask for a method and its arguments whereas the other three prompts preceed the displayed answers.

For each error occuring during the execution of a method an error message is stored by the CBserver `receiver` in the message queue of the client `sender`. This error message can be fetched via a call of method NEXT_MESSAGE, see below.

a) General Methods

ipcmessage (sender, receiver, TELL, [objects])

- objects: PROLOG string containing object descriptions in Telos seperated by '.' or '*newline*', for example: "object1.object2.object3"
- return: yes in case of success, no otherwise

The CBserver receiver checks the syntax of objects creating a parse tree for each object description, called **SMLfragment** (see appendix 2). If no syntax error occurs the SMLfragments are transformed into an internal network representation with specialized rules and constraints. Those facts which are not already retrievable are temporarily added to the KB. A check is then performed to determine whether the updated KB still satisfies the integrity constraints. In the case of satisfaction the new information is made permanent, otherwise it is deleted. If the fact UpdateMode/1 from module GlobalParameters.pro is set to persistent - its default value - all updates are stored simultanously to the PROLOG KB **and** to the files specified by the current application. See also method LOAD_APPLICATION.

Example: Telling two individual objects bill and jim as instances of the class Employee

```
Method:      @ TELL.
Arguments    ([a1,..,an]): @
      ["Individual bill in Employee with end bill.
       Individual jim in Employee with end jim"].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:    yes
```

ipcmessage (sender, receiver, TELL_OBJPROC, [[SMLfrag$_1$, ... ,SMLfrag$_n$]])

- SMLfrag$_i$: SMLfragment to be told
- return: generatedcode(listofpropvals,listofPROLOGrules,listofTimeRel), in case of failure generatedcode(nil,nil,nil)

This method allows one to tell a list of SMLfragments rather than Telos object descriptions thus avoiding one transformation step. Another difference to TELL is that the newly created internal structures are reported back.

Suppose we specified an object Tom as an instance of the class Employee:

```
Individual Tom in Employee with
      dept
            TomsDepartment: Production
      salary
            TomsSalary: 30500
end Tom
```

The call of the method TELL_OBJPROC with the corresponding SMLfragment yields:

```
Method:      @ TELL_OBJPROC.
Arguments    ([a1,..,an]): @
      [[SMLfragment( what(Tom,default,nil),
      in_omega([class(Individual,default,nil)]),
      in([class(Employee,default,nil)]),
      isa(nil), with(
      [attrdecl([dept],
      [property(TomsDepartment,Production,default,nil)]),
      attrdecl([salary],
      [property(TomsSalary,30500,default,nil)])]))]].

Responding tool:  CBserver#4703.0
Completion:       ok
Return value:     generatedcode([
proposition(Tom,Tom,-,Tom,Always,
tt(millisecond(1992,1,14,15,30,4,0))),
proposition(#4703.8,Tom,*instanceof,Employee,Always,
tt(millisecond(1992,1,14,15,30,4,0))),
proposition(#4703.9,Tom,TomsDepartment,Production,Always,
tt(millisecond(1992,1,14,15,30,4,0))),
proposition(#4703.10,#4703.9,*instanceof,#4566.9,Always,
tt(millisecond(1992,1,14,15,30,4,0))),
proposition(#4703.11,Tom,TomsSalary,35000,Always,
tt(millisecond(1992,1,14,15,30,4,0))),
proposition(#4703.12,#4703.11,*instanceof,#4566.8,Always,
tt(millisecond(1992,1,14,15,30,4,0)))],nil,nil)
```

The so called propositions proposition/6 are the basic units of the internal network representation (see section 2.1).

ipcmessage (sender, receiver, UNTELL, [objects])

- objects: PROLOG string containing object descriptions in Telos separated by '.', for example: "object1.object2.object3"
- return: yes in case of success, no otherwise

The objects in objects will be untold, i.e. the upper bound of their transaction time interval is set to the time the UNTELL operation takes place. That means from this time on the system doesn't believe this information anymore. Questions about the current state of the knowledge base yield the same answer as if the objects were never inserted into the system. However questions about earlier states will regard all information (even untold) the transaction time of which contains the time in question (= rollback time). Like in the TELL method, if the UNTELL operation would result in an inconsistent KB state it is rejected by the integrity checker. Again the updates of the UNTELL operation are written to the application files if UpdateMode is set persistent.

Example: Untelling the previously told employee jim and then querying for the existence of jim at rollback time 'Now' and at rollback time minute(1992,2,18,12,16).

```
Method: @ UNTELL.
Arguments ([a1,..,an]): @
      ["Individual jim in Employee with end jim.
      Individual jim with end jim"].

Responding tool: CBserver#4703.0
Completion:       ok
Return value:    yes
```

Note: Only untelling "`Individual Jim in Employee with end Jim`" would result in untelling the instance relationship of `Jim` to `Employee` but would not untell the object `Jim` itself.

```
Method: @ ASK.
Arguments ([a1,..,an]): @ [exists([jim/objname]),ROLLBACK:Now].

Responding tool: CBserver#4703.0
Completion:      error
Return value:    no


Method: @ ASK.
Arguments ([a1,..,an]): @
      [exists([jim/objname]),ROLLBACK:minute(1992,2,18,12,16)].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:    yes
```

ipcmessage (sender, receiver, ASK, [query, ROLLBACK:p1, FORMAT:p2])

- `query`: denotes a query according to the query language CBQL, refer to section 2.3 for details. One of the following four alternatives can be chosen:

  `queryname, deriveExp, queryFragment, querytext` where

  `queryname` denotes the name of a (generic) query class which is stored in the KB.

  `deriveExp` denotes a derived expression based on a generic query class, especially of a builtin query class, which is stored in the KB.

  `queryFragment` denotes one SMLfragment describing a query class.

  `querytext` denotes a PROLOG string containing one query object description in Telos frame syntax.

- `p1`: rollback time specification - can be omitted, then its default value is 'Now'.

- p2: optional answer format specification, possible values are: FRAGMENT, FRAME, NODE, EDGE and spezialisations of NODE and EDGE. (You can obtain further informations about the answer formats in chapter 3.2.5). Format specification can be omitted, then usually FRAGMENT is used as default value.

- return: list of answers in case of success, no otherwise

If query is a queryname or a deriveExp the named object must exist, i.e. it must have been told to the knowledge base already. The query identified by this name is evaluated. If query is a queryFragment or a querytext it is first temporarily told to the knowledge base and then evaluated. After the evaluation the query object is removed from the KB.

Example: Well_off_Manager identifies an existing query which asks for all managers whose salary exceeds a certain amount. The answer is due to the KB state of the 1. Jan 1992, no format specification.

```
Method: @ ASK.
Arguments ([a1,..,an]): @ [Well_off_Manager,ROLLBACK:day(1992,1,1)].
Responding tool:  CBserver#4703.0
Completion:       ok
Return value:
      [SMLfragment(what(Albert,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil)),
      SMLfragment(what(Hubert,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil)),
      SMLfragment(what(Konrad,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil))]
```

Example: Telling and asking a query object Still_Well_off_Manager which retrieves exactly the same answer as Well_off_Manager, no format and rollback time specification.

```
Method: @ ASK.
Arguments ([a1,..,an]): @
      ["QueryClass Still_Well_off_Manager isA Manager with
            query_rule
                  well_off_rule :   $
                                    exists s/HighSalary
                                    (this salary s)
                                    $
      end Still_Well_off_Manager " ].

Responding tool: CBserver#4703.0
Completion:       ok
Return value:     [SMLfragment(...), ...]
```

The following three queries are predefined builtin queries and available after booting the *ConceptBase* server. The first two additionally give good examples for derived expressions by instantiating parameters of generic query classes.

- exists([_x/objname])

The answer return is `yes` if there is an object named `_x`, otherwise `no`.

Example: Asking whether an object `Employee` exists.

```
Method: @ ASK.
Arguments ([a1,..,an]): @ [exists([Employee/objname])].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:    yes
```

- get_object([_x/objname])

The answer is the SMLfragment belonging to the object `_x` if there is an object `_x`. Otherwise, the answer is `no`. Only information that is explicitly stored (i.e. not inherited or deduced) is considered.

Example: Retrieving the object known to be `Tom`'s department using a select expression (see appendix 2).

```
Method: @ ASK.
Arguments ([a1,..,an]): @
[get_object([select(Tom,'!',TomsDepartment)/objname]),FORMAT:FRAGMENT].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:
      SMLfragment(what(select(Tom,!,TomsDepartment),default,nil),
      in_omega([class(Attribute,default,nil)]),
      in([class(select(Employee,!,dept),default,nil)]),
      isa(nil),with(nil))

Method: @ ASK.
Arguments ([a1,..,an]): @
[get_object([select(Tom,'!',TomsDepartment)/objname]),FORMAT:FRAME].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:
      [65,116,116,114,105,98,117,116,101,32,84,111,109,33,84,111,109,
      115,68,101,112,97,114,116,109,101,110,116,32,105,110,32,69,109,
      112,108,111,121,101,101,33,100,101,112,116,32,119,105,116,104,
      32,10,101,110,100,32,84,111,109,33,84,111,109,115,68,101,112,97,
      114,116,109,101,110,116,10]
```

The human readable form of the list is:

```
      "Atttribute Tom!TomsDepartment in Employee!dept with
      end Tom!TomsDepartment"
```

- [each, _term, where, _l1, ..., _ln]

This question provides a SQL-like query mechanism which uses the power of the literals listed in appendix 2. It is the only allowed query that is not according to the language CBQL. The item `_term` is any valid PROLOG term, `_l1` to `_ln` are literals. The answer `return` contains all terms matching `_term` which satisfy the logical conjunction of the literals.

Example: Retrieving every employee of the department `Production` and his boss.

```
Method: @ ASK.
Arguments ([a1,..,an]): @
      [[each,has_boss(_x,_y),where,
      AttrValue(_x,dept,Production,_), AttrValue(_x,boss,_y,_)]].
Responding tool: CBserver#4703.0
Completion:      ok
Return value:
      [has_boss(Hubert,Hubert), has_boss(Tom,Hubert),
      has_boss(Michael,Hubert), has_boss(Jack,Hubert),
      has_boss(Joe,Hubert), has_boss(Max,Hubert),
      has_boss(Rita,Hubert)]
```

ipcmessage (sender, receiver, HYPO_ASK, [objList, queryList, ROLLBACK:p1, FORMAT:p2])

- `objList` list of objects in fragment syntax or string of objects in frame syntax
- `queryList`: list of query names or derived expressions
- `p1`: rollback time specification, see method ASK
- `p2`: answer format specification, see method ASK
- `return`: list of answers in case of success, no otherwise

This method allows to pose so called 'hypothetical' queries against the KB. The objects in `objList` are temporarily told. This list may contain query objects which may in turn be referred to by names contained in `queryList`. Then the queries in `queryList` are evaluated as if the temporary information would belong to the KB. Afterwards the temporary information will be removed.

Example: What is the answer to the query `Well_off_Manager` if the object `Bob` would belong to the KB.

```
Method: @ HYPO_ASK.
Arguments ([a1,..,an]): @
      ["Individual Bob in Manager with salary bsalary : 115000
      dept bdept : Marketing end Bob", [Well_off_Manager]].

Responding tool: CBserver#4703.0
Completion: ok
Return value:
      [SMLfragment(what(Albert,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil)),
      SMLfragment(what(Bob,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil)),
      SMLfragment(what(Hubert,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil)),
      SMLfragment(what(Konrad,default,nil),in_omega(nil),
      in([class(Well_off_Manager,default,nil)]),isa(nil),with(nil))]
```

Note: An ordinary ASK of the query `Well_off_Manager` would not contain `Bob` in its result.

ipcmessage (sender, receiver, LOAD_APPLICATION, [path, app])

- `path`: complete path to the directory of the desired application
- `app`: name of an application
- `return`: `yes` in case of success, `no` otherwise

This method opens an application. A previously existing application is removed from the KB and the current application is set to the specified directory and application name. Files with name `app` and extension `prop` (and possibly with extensions `hprop`, `rule`, `builtin`, `_timeRel.pro`) from directory `path` are loaded into the KB if they exist, otherwise new application files are created as soon as the first object is told to the KB.
All subsequent operations are performed on this application. Especially if the fact `UpdateMode/1` from module `GlobalParameters.pro` is set to `persistent` - its default value - all updates are stored simultanously to the PROLOG KB **and** to the specified application files.

Example: Loading the `Employee` application from our sample directory.

```
Method: @ LOAD_APPLICATION.
Arguments ([a1,..,an]): @
      ['$CB_HOME/CB_Exe/Examples/QUERIES/', Employee].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:    yes
```

ipcmessage (sender, receiver, NEXT_MESSAGE, [ ])

- `return`: contains the next message for the client if its message queue contains at least one message, `empty_queue` if no message exists

Client `sender` requests a message from the CBserver `receiver` stored in its message queue.

Example: Suppose an error occurred telling that employee Tom earns more than his boss. The corresponding error message can be retrieved by using NEXT_MESSAGE:

```
Method: @ NEXT_MESSAGE.
Arguments ([a1,..,an]): @ [].

Responding tool: CBserver#4703.0
Completion:      ok
Return value:
      ipcmessage(
      CBserver#4703.0,
      ExampleIpcClient#4703.1,
      ERROR_REPORT,
      [Error BDMChIcA in module BDMCheck.,
      [The  literal  AttrValue(Tom,boss,Hubert,Always)  violates  the,
      integrity constraint #4566.32.]])
```

b) Privileged Methods

These methods affect other clients connected to CBserver as well and should only be executed by an authorized client.

`ipcmessage ( sender, receiver, STOP_SERVER, [password] )`

- `password`: password allowing a client to stop a CBserver
- `return`: `yes` in case of success, `no` otherwise

The CBserver `receiver` is terminated (PROLOG process stopped) if the password is correct and the user running the client is also the owner of the CBserver to be stopped. To the requesting client STOP_SERVER has the same effect as CANCEL_ME. It is recommended to terminate the CBserver by using the respective menu choice from the "System Menu" of *ConceptBase* UsageEnvironment if you want to stop the CBserver process.

`ipcmessage ( sender, receiver, REPORT_CLIENTS, [ ] )`

- `return`: list of all clients currently connected to CBserver `receiver`

CBserver `receiver` reports back the identifier, toolclass and owner name of all currently connected clients including itself.

`ipcmessage ( sender, receiver, REPORT_TOOLCLASS, [toolclass] )`

- `toolclass`: contains the name of a tool class, e.g. 'ExampleIpcClient'
- `return`: list of clients of requested tool class , no otherwise

This method reports all clients currently connected to CBserver `receiver` that belong to the same tool class `toolclass`.

Example: Asking for all clients with tool class `ExampleIpcClient` currently connected to CBserver.

```
Method: @ REPORT_TOOLCLASS.
Arguments ([a1,..,an]): @ [ExampleIpcClient].

Responding tool: CBserver#4703.0
Completion:        ok
Return value:   [tool(ExampleIpcClient#4703.1,example_user),
      tool(ExampleIpcClient#4703.13,example_user),
      tool(ExampleIpcClient#4703.14,example_user)]
```

c) Internal Methods

These methods are for internal purposes only. A client should use the corresponding procedures the programming environment exports (see next section).

`ipcmessage ( sender, receiver, ENROLL_ME, [toolclass,username] )`

- `toolclass`: 'class' the client belongs to
- `username`: name of the user running the client
- `return`: identifier assigned to the client by CBserver

`sender` and `receiver` have value '_' since they are not known. The sending client will be registered as a new client of the CBserver with its own identifier and message queue. This message must be sent to the CBserver before any other message can be sent, since all other messages require valid identifiers to be assigned to sender and receiver. It is recommended to use `connectCBserver`/2 which implicitly performs an enrollment.

`ipcmessage ( sender, receiver, CANCEL_ME, [ ] )`

- `return`: `yes` in case of successful disconnection, `no` otherwise

Client `sender` will be disconnected from CBserver `receiver`. This means that from now on the sender is no longer known to the CBserver (no further messages can be sent) and its message queue is deleted. It is recommended to `use disconnectCBserver/0` which implicitly cancels a connection.

`ipcmessage ( sender, receiver, PROLOG_CALL, [call] )`

- `call`: any PROLOG predicate to be executed by CBserver
- `return`: `yes` if `call` can be proven, `no` otherwise

This is for debugging and testing purposes only.

abc

References

[BBD*90] Bellosta, M.J., Bessede, A., Darrieumerlou, C., Gruber, O., Pucheral, P., Thevenin, J.M., Steffen, H. (1990). GEODE: Concepts and Facilities. Manual, INRIA, Roquencourt, France.

[BBMR89] Borgida, A., Brachman, R.J., McGuiness, D.L., Resnick, L.A. (1989). CLASSIC: a structural data model for objects. Proc. ACM-SIGMOD Intl. Conf. Management of Data, Portland, Or, 58-67.

[BCG*87] Banerjee, J., Chou, H.-T., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H.-J. (1987). Data model issues for object-oriened applications. ACM Trans. Office Information Systems 5, 1, 3-26.

[BIM90] BIM SA/NV (1990). ProLog by BIM, reference manual for release 3.0. Update for release 3.1 (1992). Everberg, Belgium.

[CHEN76] Chen, P. P.-C. (1976). The Entity-Relationship-Model - Towards a Unified View of Data. ACM Transact. on Database Systems. Vol. 1, No. 1, March 1976, pp. 9-36.

[EJJ*89] Eherer, S., Jarke, M., Jeusfeld, M., Miethsam, A., Rose, T. (1989). ConceptBase V2.0 user manual. Report MIP-8936, Universität Passau, Germany.

[HJEK90] Hahn, U., Jarke, M., Eherer, S., Kreplin, K. (1990). CoAUTHOR: a hypermedia group authoring environment. In Benford, J.M., Bowers, S.D. (eds.): Studies in Computer-Supported Cooperative Work, North-Holland, 79-100.

[JEUS92] Änderungskontrolle in deduktiven Objektbanken. Dissertation, Universität Passau, Germany.

[JJ91] Jarke, M., Jeusfeld, M. (1991). From Relational to Object-Oriented Integrity Simplification. *Proc. 2nd Intl. Conf. Deductive and Object-Oriented Databases*, Munich, Dec. 1991; also as *Aachener Informatik-Berichte* 91-19, RWTH Aachen, Germany.

[JJR88] Jarke, M., Jeusfeld, M., Rose, T. (1988). A KBMS for database software evolution: documentation of first ConceptBase prototype, Report MIP-8819, Universität Passau, Germany.

[JJR90] Jarke, M., Jeusfeld, M., Rose, T. (1990). A software process data model for knowledge engineering in information systems. *Information Systems 15, 1,* 85-116.

[JMSV90] Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y. (1990). Information systems development as knowledge engineering: a review of the DAIDA project. *Programirovanie* 17(1), Academy of Sciences, Russia; also report MIP-9010, Universität Passau, Germany.

[JS92] Jeusfeld,M., Staudt,M. (1992). Query optimization in deductive object bases. To appear in Freytag, Vossen, Maier (eds.): Query Processing for Advanced Database Applications, Morgan-Kaufmann, 1992; also as *Aachener Informatik-Berichte* 91-26, RWTH Aachen, Germany.

[MBJK90] Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M. (1990). Telos: a language for representing knowledge about information systems. *ACM Trans. Information Systems* 8, 4.

[RD92] Ramesh,B., Dhar,V. (1992). Process knowledge-based group support for requirements engineering. *Journal Intelligent Information Systems* 1, 1.

[RJG*91] Rose, T., Jarke, M., Gocek, M., Maltzahn, C., Nissen, H. (1991). A decision-based configuration process environment. *Software Engineering Journal*, Special Issue on Software Environments and Factories, July 1991

[STAU90] Staudt, M. (1990). Anfragerepräsentation und -auswertung in deduktiven Objektbanken. Diploma thesis, Universität Passau.

[WING90] Wing, Jeanette M. (1990). A Specifier's Introduction to Formal Methods. *IEEE Computer*, No. 9, pp 8-24, Sept. 1990.

abc

Appendix 1a: Installation Guide for the runtime version of CB

===========

ConceptBase

V3.1

===========

Installation Guide (Runtime version)

Martin Staudt
RWTH Aachen
15-Nov-1992

Abstract. This is the installation guide for the runtime version of *ConceptBase V3.1*. It should be used if no full development license for ProLog by BIM exists. Besides instructions for installing *ConceptBase* it contains the necessary steps for starting the knowledge base server and the usage environment.

CONTENTS

1. Requirements and Preparations

Computer:	SUN4/Sparc
Main memory:	8 MB (16 recommended)
Free swap space:	>= 30 MB (40 MB recommended)
Free space on hard disk:	>= 16 MB
Operating system:	Sun UNIX, Release 4.1.1
Environment:	X11 R5

We assume that your machine conforms to the above requirements. Insert in your file `~/.cshrc` the following commands

```
setenv CB_HOME <dir_of_CB>
setenv CB_RUNTIME_DIR $CB_HOME/CB_Exe/KSV_Exe
setenv BIM_PROLOG_DIR $CB_RUNTIME_DIR
```

where <dir_of_CB> is the installation directory of *ConceptBase* (see below). Any user of *ConceptBase* must specify these lines in his `~/.cshrc`.

Then execute in a C-Shell (`/bin/csh`):

```
% source ~/.cshrc
```

We strongly recommend that you install a group 'cbase' in `/etc/group` and assign each *ConceptBase* user to it. In addition an `umask` setting is necessary which allows other CB users to read and write commonly used application files.

2. Loading the system from tape

Change to the directory where you want to install *ConceptBase*.

```
% cd <dir_of_CB>
```

Next, insert the tape in a 150 MB streamer tape drive and extract *ConceptBase* by

```
% tar -xvfb /dev/rst0 126
```

The directory `CB_Exe` (which contains all *ConceptBase* files in subdirectories `KSV_Exe` and `XI_Exe`) will be copied as a subdirectory in your directory <dir_of_CB>. Then run the following command in `CB_Exe/KSV_Exe`:

```
% helloConceptBase
```

If you have set up a group `cbase` (see above) then assign group tag to the *ConceptBase* files:

```
% cd $CB_HOME
% chgrp -R cbase CB_Exe
```

If you don't want to install a group `cbase` then only the owner of the `$CB_HOME` directory can use *ConceptBase*.

3. Starting the ConceptBase Server

You have to use a C-Shell (`/bin/csh`) to start *ConceptBase*. Start the server by

```
% $CB_HOME/CB_Exe/KSV_Exe/runCBserver -p 4001
```

The number `4001` is the port (or "socket") number that is used to connect clients to the server process. You can specify any number between `3000` and `8000`. The default value is `4001` (if you ommit the '`-p`' option). After successful loading the server reports

```
This is ConceptBase (CBserver) ...
```

with some additional information.

Please note that no two servers can use the same port number on the same machine. In this case you get the error message

```
IPC Error: Unable to bind socket to name
```

and the server won't start up. You should then either start the server on a different machine or use another port number.

Other command line parameters are

`-t <trace>` This parameter controls the amount of tracing on the CBserver terminal. Possible settings are:
`none`: (nearly) no tracing
`low`: only interface calls (default)
`high`: intermediate steps of transformations
`veryhigh`: many-many-many lines of output

`-u <upd>` Set `<upd>` to `nonpersistent` if your updates shall be valid only for the current session. Otherwise, set it to `persistent` (default).

`-q <quev>` Queries and deductive rules can be evaluated backwardly (`SLDNF`) by Prolog-like resolution and "forwardly" (`Magic`). We recommend `SLDNF` (default) (see 2.3).

`-o <opt>` The optimization of integrity constraints by using structural axioms can be turned on (`structural`) or off (`noopt`).

Specified parameters must be separated by blanks.

4. Starting the ConceptBase Usage Environment

Start the X11 usage environment of *ConceptBase* in a C-Shell by

```
% $CB_HOME/CB_Exe/XI_Exe/sun4/CBinterface
```

Successful starting is indicated by the appearing *ConceptBase Tool Bar*.

5. Connecting Server and Usage Environment

After starting server and usage environment of CB the first step is to establish the connection between them. This is done by selecting menu item

```
Connect CB server
```

from the `System` menu of the tool bar. In the appearing interaction window the host name and port number of the CBserver must be specified and as a result the connection is established.

Example:

1. start server on machine `alpha` with port number `5000` (see above)
2. start usage environment on machine `beta` (see above)
3. connect usage environment on `beta` to the server on `alpha` by selecting `Connect CB server` and specifying

```
host    : alpha
port nr : 5000
```

Whenever a new connection from a usage environment to a CBserver is created a possibly existing 'old' one is disconnected. Please note that the machines `alpha` and `beta` can be located anywhere in the *Internet*. You can also start both processes on the same machine.

6. Additional remarks

A CBserver may only be stopped by a client if the name of the user of the client is the same as the one who started the CBserver. Use menu item `Report clients` of the usage environment to display the user names. This behavior is not an error but a security measure.
The communication between the client tools and the error window is based on the UNIX IPC message service. Hence, the kernel of each machine has to be configured with the IPCMESSAGE option set. Otherwise, errors will be lost and not reported.

abc

Appendix 1b: Installation Guide for the PROLOG version of CB

===========

ConceptBase

V3.1

===========

Installation Guide

Martin Staudt
RWTH Aachen
15-Nov-1992

Abstract. This is the installation guide for the interpreter version of *ConceptBase V3.1*. It should be used if a full development license for ProLog by BIM is available. Besides instructions for installing *ConceptBase* it contains the necessary steps for starting the knowledge base server and the usage environment.

CONTENTS

1. Requirements and Preparations

Computer:	SUN4/Sparc
Main memory:	8 MB (16 recommended)
Free swap space:	>= 30 MB (40 MB recommended)
Free space on hard disk:	>= 16 MB
Operating system:	Sun UNIX, Release 4.1.1
Programming language:	ProLog by BIM V3.1, C
Environment:	X11 R5

We assume that your machine conforms to the above requirements. Make sure that you have installed ProLog by BIM, and insert in your file `~/.cshrc` the following commands

```
setenv CB_HOME <dir_of_CB>
setenv BIM_PROLOG_DIR <dir_of_BIM_Prolog>
```

where `<dir_of_CB>` is the installation directory of *ConceptBase* (see below), `<dir_of_BIM_Prolog>` is the installation directory of ProLog by BIM (see installation hints of BIM). Any user of *ConceptBase* must specify these lines in his `~/.cshrc`.

Then execute in a C-Shell (`/bin/csh`):

```
% source ~/.cshrc
```

We strongly recommend that you install a group 'cbase' in `/etc/group` and assign each *ConceptBase* user to it. In addition an `umask` setting is necessary which allows other CB users to read and write .pro and .wic files of ConceptBase as well as commonly used application files.

2. Loading the system from tape

Change to the directory where you want to install *ConceptBase*.

```
% cd <dir_of_CB>
```

Next, insert the tape in a 150 MB streamer tape drive and extract *ConceptBase* by

```
% tar -xvfb /dev/rst0 126
```

The directory `CB_Exe` (which contains all *ConceptBase* files in subdirectories `KSV_Exe` and `XI_Exe`) will be copied as a subdirectory in your directory `<dir_of_CB>`. Then run the following command in `CB_Exe/KSV_Exe`:

```
% helloConceptBase
```

If you have set up a group `cbase` (see above) then assign group tag to the *ConceptBase* files:

```
% cd $CB_HOME
% chgrp -R cbase CB_Exe
```

If you don't want to install a group `cbase` then only the owner of the `$CB_HOME` directory can use *ConceptBase*.

3. Starting the ConceptBase Server

Before you start the ConceptBase server please check wether you have installed the right ProLog by BIM Release. You find the necessary release specification among all other installation specific settings in the file

`$CB_HOME/CB_Exe/KSV_Exe/GlobalParameters.pro`

as arguments of a fact `cb_prologcompiler/2`.
If you have a different version then *ConceptBase* will NOT run properly. In this case please ask your BIM representative for the appropriate version. Our policy is to move as soon as possible to new releases of ProLog by BIM.
Other installation specific parameter in `GlobalParameters.pro` are:

```
TraceMode/1
RangeFormOptimizing/1
UpdateMode/1
QueryRep/1
```

You find explanations for these parameters in `GlobalParameters.pro` and you can modify their values according to your preferences. Don't change any other parts of this file !

Use a C-Shell (`/bin/csh`) to start the *ConceptBase* server by

```
% $CB_HOME/CB_Exe/KSV_Exe/goCBserver 4001
```

The number `4001` is the port (or "socket") number that is used to connect clients to the server process. You can specify any number between `3000` and `8000`. The default value is `4001` (if you ommit this argument). After successful loading the server reports

```
This is ConceptBase (CBserver) ...
```

with some additional information.

Please note that no two servers can use the same port number on the same machine. In this case you get the error message

```
IPC Error: Unable to bind socket to name
```

and the server won't start up. You should then either start the server on a different machine or use another port number.

4. Starting the ConceptBase Usage Environment

Start the X11 usage environment of *ConceptBase* in a C-Shell by

```
% $CB_HOME/CB_Exe/XI_Exe/sun4/CBinterface
```

Successful starting is indicated by the appearing *ConceptBase Tool Bar*.

5. Connecting Server and Usage Environment

After starting server and usage environment of CB the first step is to establish the connection between them. This is done by selecting menu item

```
Connect CB server
```

from the `System` menu of the tool bar. In the appearing interaction window the host name and port number of the CBserver must be specified and as a result the connection is established.

Example:

1. start server on machine `alpha` with port number `5000` (see above)
2. start usage environment on machine `beta` (see above)
3. connect usage environment on `beta` to the server on `alpha` by selecting `Connect CB server` and specifying

```
host    : alpha
port nr : 5000
```

Whenever a new connection from a usage environment to a CBserver is created a possibly existing 'old' one is disconnected. Please note that the machines `alpha` and `beta` can be located anywhere in the *Internet*. You can also start both processes on the same machine.

6. Additional remarks

A CB server may only be stopped by a client if the name of the user of the client is the same as the one who started the CBserver. Use menu item `Show/Report clients` of the usage environment to display the user names. This behavior is not an error but a security measure.

The communication between the client tools and the error window is based on the UNIX IPC message service. Hence, the kernel of each machine has to be configured with the IPCMESSAGE option set. Otherwise, errors will be lost and not reported.

abcd

Appendix 2: Syntax Specifications

A2.1 Syntax Specifications for Telos Objects

```
<object>             ::=  [<classList>] <objectIdentifier> <objectSpec>
                          end [<objectIdentifier>]
<objectSpec>         ::=  [in <classList>] [isA <classList>]
                          [with [<AttrDeclList>]]
<classList>          ::=  <objectIdentifier>,<classList> |
                          <objectIdentifier>
<AttrDeclList>       ::=  <attrCategoryList> <AttrDecl> <AttrDeclList> |
                          <attrCategoryList> <AttrDecl>
<AttrDecl>           ::=  <Label> : <objectIdentifier> ; <AttrDecl> |
                          <Label> : <objectIdentifier>
<Label>              ::=  <label> | ε
<attrCategoryList>   ::=  <attrCategory> , <attrCategoryList> |
                          <attrCategory>
<attrCategory>       ::=  Identifier
<objectIdentifier>   ::=  <objectIdentifier> <selector> <label> |
                          Identifier
<selector>           ::=  ! | => | -> | ^
<label>              ::=  Identifier
```

Functors are built in the same way as functors in BIMProlog, identifiers in the same way as atoms. Figure A2-1 shows a diagram of Telos syntax. Rectangles mean nonterminal, ovals terminal symbols. Lines without arrows should be read left to right.

A2.2 Syntax of the Assertion Language

```
<AssertionString>    ::=  <ruleString> | <constraintString> |
                          <queryRuleString>
<ruleString>         ::=  forall <variableBind> (<formula>) ==> <literal>|
                          <formula> ==> <literal>
<constraintString>   ::=  <formula>
<queryRuleString>    ::=  <formula>
<formula>            ::=  exists <variableBind> <formula> | <literal> |
                          not <formula> | forall <variableBind> <formula>|
                          <formula> ==> <formula> |
                          <literal> and <formula> |
                          <literal> or <formula> | ( <formula> )
<variableBind>       ::=  <var> / <class> <variableBind> | <var> / <class>
<literal>            ::=  <functor> ( <literalArgList> ) |
                          ( <infixLiteral> ) | <infixLiteral>
<literalArgList>     ::=  <literalArg> , <literalArgList> | <literalArg>
<infixLiteral>       ::=  <literalArg> <infixSymbol> <literalArg>
<literalArg>         ::=  Identifier | <deriveExp> | <selectExp>
<selectExp>          ::=  <selectExp> <selector> Identifier | Identifier
```

```
<selector>          ::= ! | => | -> | ^
<infixSymbol>       ::= < | > | <= | >= | = | <> | == | in | isA |<label>
<functor>           ::= AttrId | LT | GT | LE | GE | EQ | NE |IDENTICAL|
                        From | Label | To | InstanceOf | IsA | AttrValue
<deriveExp>         ::= Identifier([<deriveExpList>]) |
                        Identifier[<deriveExpList>]
<deriveExpList>     ::= <singleExp> , <deriveExpList> | <singleExp>
<singleExp>         ::= Identifier / Identifier | Identifier : Identifier
<label>             ::= Identifier
<var>               ::= Identifier
```

Fig. A2-2 shows a diagram of Telos' assertion language. See Chapter 2 for integrating AssertionStrings into Telos objects.

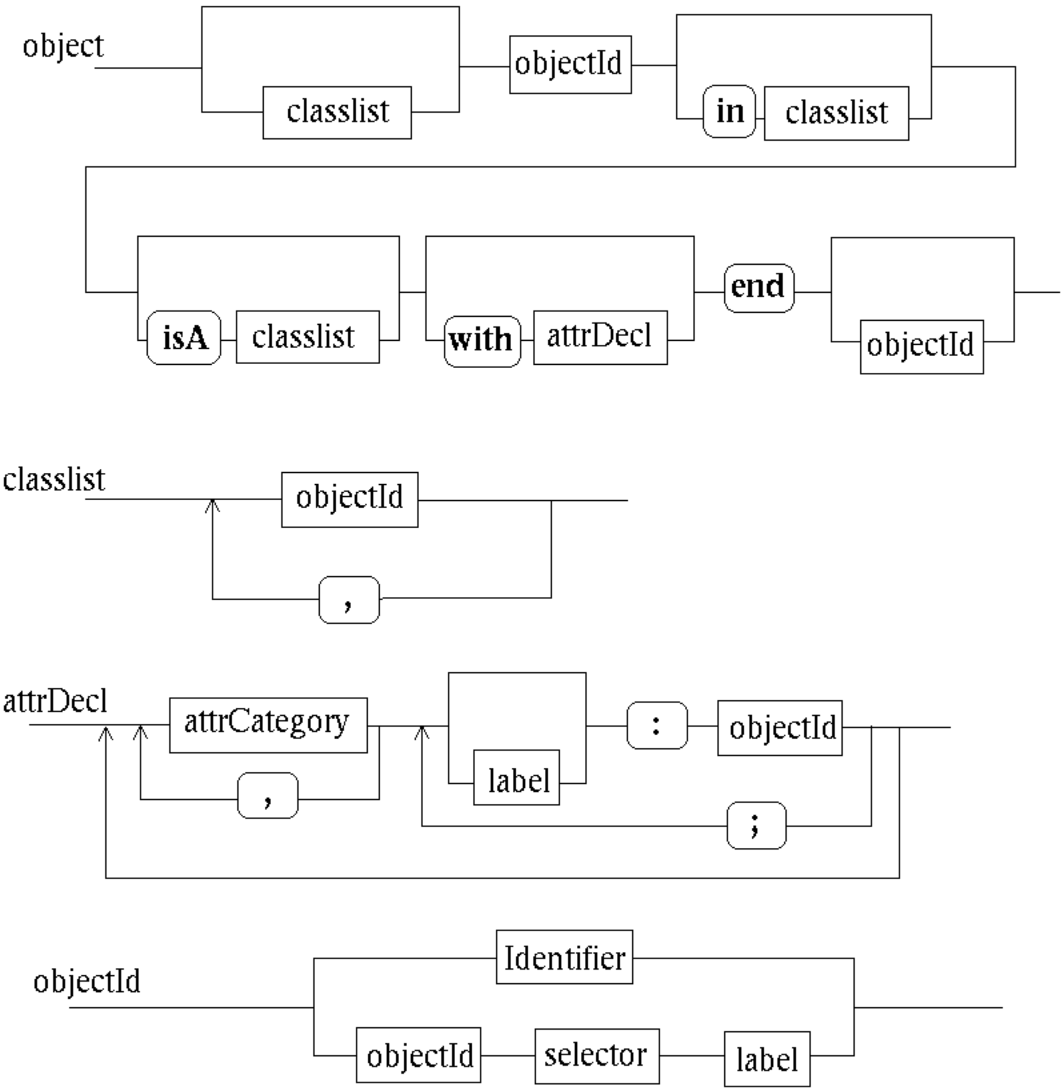


Fig. A2-1: Telos syntax diagram

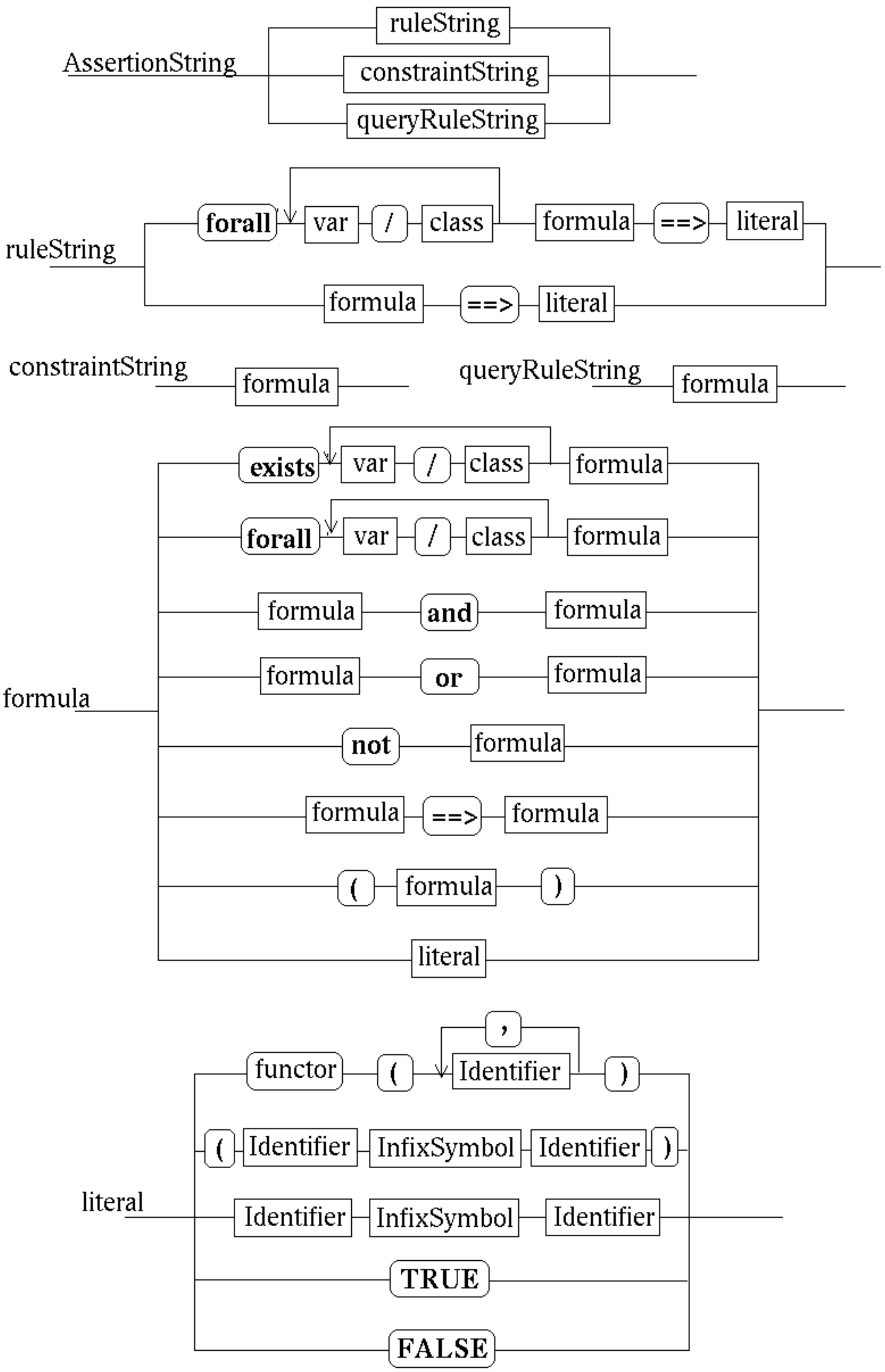


Fig. A2-2: Syntax Diagram of *ConceptBase*' Assertion Language

A2.3 Syntax Specifications for SML Fragments

```
<SMLfragment>        ::=  SMLfragment(<what>,<in_omega>,<in>,<isa>,<with>)
<what>               ::=  what(Identifier,<time_identifier>,
<timerellist>)
<in_omega>           ::=  in_omega(<classlist>)
<in>                 ::=  in(<classlist>)
<isa>                ::=  isa(<classlist>)
<with>               ::=  with(<attrdecllist >)
<classlist>          ::=  [] | [<class> {,<class>}]
<class>              ::=  class(Identifier      ,      <time_identifier>,
<timerellist>)
<attrdecllist>       ::=  [] | [<attrdecl> {,<attrdecl>}]
<attrdecl>           ::=  attrdecl(<attrcategorylist>, <propertylist>)
<attrcategorylist>   ::=  [] | [<attrcategory> {,<attrcategory>}]
<propertylist>       ::=  [] | [<property> {,<property>}]
<property>           ::=  property(Identifier, Identifier ,
                          <time_identifier>, <timerellist> )
<attrcategory>       ::=  Identifier
<timerellist>        ::=  []
<time_identifier>    ::=  Always
```

abc

Appendix 3: Examples

A3.1 Example Model: The Employee Model

```
{*********************************************************************}
{        File : Employee_Classes.sml                                  }
{        created : 9/18/89 SE                                         }
{        last change : 13-Jun-1992/HWN                                }
{*********************************************************************}


Class Money in SimpleClass with end Money
Class Integer isA Money with end Integer

Class Employee in SimpleClass with
   attribute
      name : String
      salary : Money
      dept : Department
      boss : Manager
end Employee

Class Manager in SimpleClass isA Employee with end Manager

Class Department in SimpleClass with
   attribute
      head : Manager
end Department

Class Employee with
   rule, bdmrule
      bossrule :
      $ forall e/Employee  m/Manager
      ((exists d/Department
        (e dept d) and
        (d head m)
       )
       ==> (e boss m)
      ) $

   bdmconstraint
      SalaryBound :
      $ forall e/Employee b/Manager
      x/Integer y/Integer
      (( (e boss b) and
         (e salary x) and
         (b salary y)
       )
       ==> (y >= x)
      ) $

end
```

```
{*****************************************************************}
{        File : Employee_Instances.sml                            }
{        created : 9/18/89 SE                                     }
{        last change : 9/18/89 SE                                 }
{*****************************************************************}

{       Instances of the class Department       }

Individual Production in Department with
   head
      Boss_of_Production : Hubert
end Production

... etc.


{       Instances of the class Manager       }

Individual Hubert in Manager with
   dept
      HubertsDepartment : Production
   salary
      HubertsSalary : 100000
end Hubert

... etc.


{       Instances of the class Employee       }

Individual Michael in Employee with
   dept
      MichaelsDepartment : Production
   salary
      MichaelsSalary : 30000
end Michael

Individual Jack in Employee with
   dept
      JacksDepartment : Production
   salary
      JacksSalary : 30500
end Jack

Individual Joe in Employee with
   dept
      JoesDepartment : Production
   salary
      JoesSalary : 35000
end Joe
```

```
{*****************************************************************}
{     File : Employee_Queries.sml                                 }
{     created : 9/18/89 SE                                        }
{                                                                 }
{*****************************************************************}

{*****************************************************************}
{                        Q u e r y   1                            }
{                                                                 }
{      ask for all Managers who are social interested             }
{                                                                 }
{*****************************************************************}

QueryClass SI_Manager isA Manager,UnionMember with
   retrieved_attribute
      union : Union;
      salary : Money
end SI_Manager

{*****************************************************************}
{                        Q u e r y   2                            }
{                                                                 }
{      ask for all Managers who have a high salary                }
{                                                                 }
{*****************************************************************}

QueryClass Well_off_Manager isA Manager with
   query_rule
      well_off_rule : $ exists s/HighSalary
                        (this salary s) $
end Well_off_Manager

{*****************************************************************}
{                        Q u e r y   3                            }
{                                                                 }
{      ask for all Managers who are social interested and for the }
{      Department they are the head of                            }
{                                                                 }
{*****************************************************************}

QueryClass Well_off_SI_Manager3 isA Manager,UnionMember with
   retrieved_attribute
      union : Union
   computed_attribute
      head_of : Department
   query_rule
      well_off_head_of_rule : $ exists s/HighSalary
                                (this salary s) and
                                (head_of head this) $
end Well_off_SI_Manager3

... etc.
```

abcd

A 3.2 A Telos Modeling Example - ER Diagrams

1. Introduction

This chapter illustrates some Telos features with the modeling of entity-relationship diagrams. The purpose of this example is to support an intuitive understanding of the concepts the generalization, classification, and aggregation dimension, metaclasses, rules and constraints, also to illustrate the adaption of the graph browser to specialized models. There is no claim that the model fully and faithfully represents all aspects of ER. In fact, we had this exercise purposefully carried out and described by students with no previous exposure to Telos.

2. Characteristics of ER Diagrams

Entity-relationship (ER) diagrams were defined as a semi-formal method for data modeling [Chen76]. His definition has been commonly used until today with only slight modifications and enhancements. For our example the following properties of the ER concept are used:

- each ER model consists of three basic components: entities, attributes, and relationships
- an entity is an object which can be distinctly identified (i.e. a specific person, a company, or an event)
- a relationship is an association among entities
- an attribute specifies properties of an entity

In a more mathematical sense:
- entities can be described as sets E_1, .. ,E_n
- relationships are subsets of the cartesian product of j (j>=1) of these sets E_i
- attributes are functions from an entity E_i to a value set

Several important characteristics of ER diagrams:
- an ER diagram can be seen as a graph where entities, relationships, and attributes are the nodes linked by edges with some limitations:
  (i) entities are linked only with relationships and attributes
  (ii) relationships are linked only with one or more entities
  (iii) attributes are linked with exactly one entity
- an ER diagram can express the *existence dependency* of one entity type on another (such dependent entities are represented by a special rectangular box)
- the diagram can distinguish between 1:n, m:n, and 1:1 mappings. This information about the number of entities e_i in each E_j which is allowed in a relationship is indicated by specifying a number interval within the positive Cardinal-range in the diagram.

The objective of the following modeling phase is to represent these characteristics in Telos.

3. The Modeling Phase

The internal hierarchical structure of ER diagrams suggests a subdivision of the implementation into three levels. ER diagram and ER objects are on the highest level (metametalevel) because of their abstractness. Entities, relationships, and attributes as the formulation of the more abstract ER objects, are one level below (metalevel). On this level, details and characteristics of the ER diagram are established. Finally, specific ER diagram are implemented on the lowest level (class level). This section deals first with the first two levels, then a part of the well-known example of the library system [Wing90] as a specific ER diagram will be presented.

3.1 Metameta Level

On this most abstract level of the following ER diagram modeling, the formulation of the 'keyword sentence' above is adopted word by word and implemented in TELOS:

```
Individual ERdiagram in Class with
attribute
        consistsOf : ERobjects
end ERdiagram

Individual ERobjects in Class with
end ERobjects
```

All of the following refinements (i.e. entities, attributes, relationships) are instances of the individual `ERobjects`.

3.2 Meta Level

Entities represent the center of the ER concept used by attributes and relationships.

```
Individual Entity in ERobjects
end Entity
```

Existence dependency was mentioned as an important feature, which can be expressed by ER diagrams. To realize this feature, an individual `DepEntity` is introduced which comprises the information about the entities on which `DepEntity` depends.

```
Individual DepEntity in ERobjects isA Entity with
necessary
        depOn  : Entity
end DepEntity
```

`DepEntity` is a specialization (isA) of `Entity` with added attributes. The attribute metaclass `necessary` forces the user to define 1..n entities, on which `DepEntity` depends. The attribute metaclasses `necessary` (1..n) and `single` (0..1) [MBJK90] are often used because they offer the possibility to limit the number of attribute-values a user can define and still have a relatively simple logical fpormaization which can be efficiently evaluated [JEUS92].
Next, the attributes have to be designed in Telos. Since `attribute` is a keyword in Telos, the identifier `Feature` is used for ER attributes.
A first attempt to model ER attributes would be :

```
Individual  Feature in ERobjects with
attribute
        belongsTo : Entity
end Feature
```

An ER attribute belongs to the specific entity, on which it is defined as a function. To check the correctness of such a definition, it is important to simulate different test cases. First, it is possible to define an attribute which belongs to zero entities. This must be a mistake, so the attribute metaclass `necessary` has to be added to guarantee that the user indicates at least 1..n entities. But defining n entities must be a mistake, too. Each attribute has to be linked with exactly one entity. The attribute metaclass `single` prevents such a wrong definition, and the combination of `single` and `necessary` offers a concise translation of 'exactly one'. Thus we get:

```
Individual Feature in ERobjects with
single, necessary
       belongsTo  : Entity
end Feature
```

A relationship is linked to 1..n entities. An example for further clarification:

- the relationship `marriage` has two links to the same entity `person`, which is not subdivided into male and female person.
- the trivial relationship `unemployed` has one link to the entity `employee` (this type of relationships often describe characteristics of entities)

To summarize the two examples, it is possible to link a relationship with 1..n entities. Thus, constraints in defining a relationship with its used entities concern the number of links, not the number of different entities:

```
Individual Relationship in ERobjects with
necessary
       needsEntities  : Entity
end Relationship
```

Referring to the marriage example above the relationship `marriage` in Telos would have two links to two entities with the same identifier `person`.
ER diagrams have the ability to express 1:1, 1:n, and m:n mappings as symbols, which belong to the links between relationship and entities. The realization of these symbols takes place at the simple class level because of the semantic effects of the mapping information on an instantiated ER diagram at the token level. To continue the marriage example, the semantical information of the 1:1 mapping could be realized using the combination of `single` and `necessary` on level 3:

```
Individual marriage in Relationship with
needsEntities, single, necessary
       partner1  :  person;
       partner2  :  person
end marriage
```

A m:n mapping is realized by using `necessary` only, a 1:m mapping by using `single` and `necessary` in the first case and `necessary` in the second case. To offer the user a range, within the number n can vary, we define a minimum and maximum number on the metalevel. But still one problem remains: There is no predefined class representing positive numbers. This difficulty is solved by defining a `card` constraint, which restricts the possible values of `numb`, allowing only positive values:

```
Attribute Relationship!needsEntities in Class, Necessary with
single, necessary
       min  :  Integer;
       max  :  Integer
bdmconstraint
       cardmin  :     $       forall x/Relationship!needsEntities
                              forall i/Integer ((x min i) ==> (i>0)) $;
       cardmax  :     $       forall x/Relationship!needsEntities
                              forall i/Integer ((x max i) ==> (i>0))  $
end Relationship!needsEntities
```

With this last definition the modeling of the ER diagram is finished. To offer a better overall view, the entire ER diagram realization in Telos is shown below:

```
Individual ERdiagram in Class with
attribute
       consistsOf  : ERobjects
end ERdiagram

Individual ERobjects in Class
end ERobjects

Individual Entity in ERobjects
end Entity

Individual DepEntity in Class isA Entity
necessary
       depOn  :  Entity
end DepEntity

Individual Feature in ERobjects with
single, necessary
       belongsTo  :  Entity
end Feature

Individual Relationship in ERobjects with
necessary
       needsEntities  :  Entity
end Relationship

Attribute Relationship!needsEntities in Necessary with
single, necessary
       min  :  Integer;
       max  :  Integer
bdmconstraint
       cardmin  :     $      forall x/Relationship!needsEntities
                             forall i/Integer ((x min i) ==> (i>0)) $;
       cardmax  :     $      forall x/Relationship!needsEntities
                             forall i/Integer ((x max i) ==> (i>0))  $
end Relationship!needsEntities
```

The next figure shows a graphical representation of the Telos definition above displayed by the ConceptBase GraphBrowser with its standard graphical type conventions, using different shades of gray to represent the different meta levels.

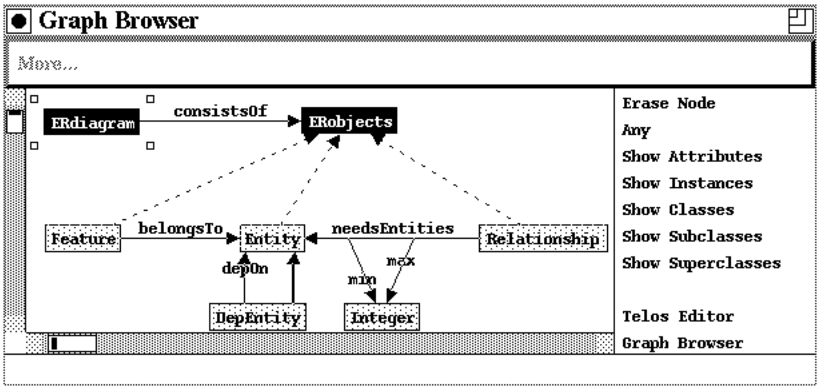


Fig. A 3.2-1: graphical representation of the ER modeling in TELOS

To clarify the use of this ER diagram definition, the next section offers a partial realization of the library system example [Wing90] .

4. The library system as an example for an ER diagram realization

The following three figures display the partial development of the library system example. Fig. A 3.2-2 shows some aspects of the library system realized in an ER diagram, Fig. A 3.2-3 the definition in Telos of the marked area of the ER diagram, and Fig. A 3.2-4 the graphical representation of the Telos frames with the realization of the library system.

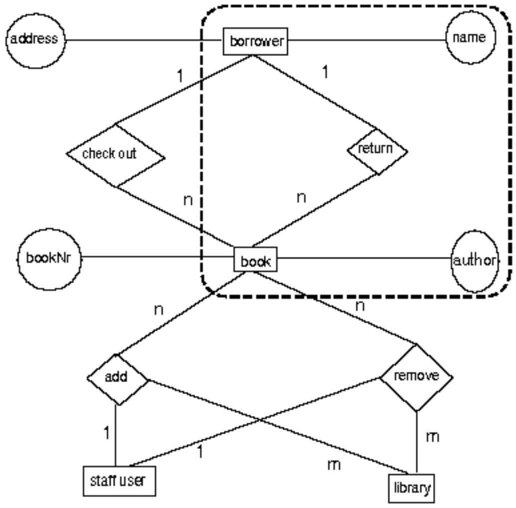


Fig. A 3.2-2: some aspects of the library system modeled as an ER diagram

```
Individual borrower in Entity with
end name

Individual book in Entity with
end book

Individual  name in Feature with
belongsTo
       identifies  :  borrower
end name

Individual author in Feature with
belongsTo
       identifies  :  book
end author

Individual return in Relationship with
needsEntities, necessary
       en1 : book
needsEntities, necessary, single
       en2 : borrower
end return

Attribute return!en1 in Relationship!needsEntities with
min
       min1  : 1
max
       max1  : 10
end return!en1

Attribute return!en2 in Relationship!needsEntities with
min
       min2  :  1
max
       max2  :  1
end return!en2
```

Fig. A 3.2-3: partial realization in Telos of the ER diagram above

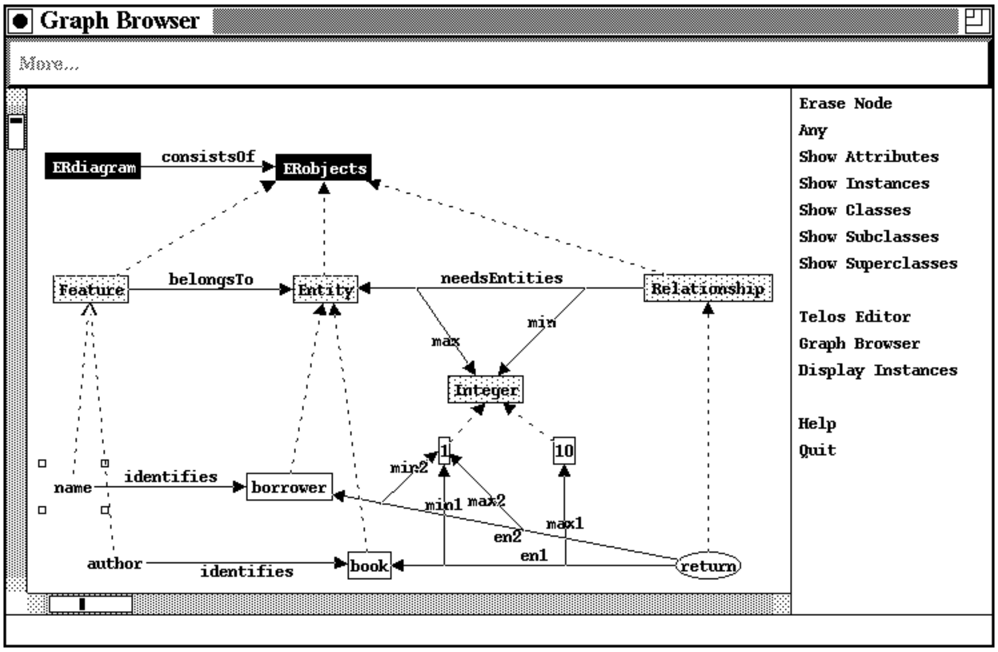


Fig. A 3.2-4: graphical representation of the TELOS definitions above

Fig. A 3.2-4 shows an example for an application specific configuration of the GraphBrowser as explained in section 3.2.5 The Objects `ERdiagram` and `ERobjects` at the top level are denoted as black rectangles, the components of an ER diagram `Entity`, `Relationship`, and `Feature` as dotted rectangles. Concrete instances of `Relationship` are displayed as white oval nodes, entities as white rectangles (default), and features of entities as labels without surrounding boxes. This can be reached by the following format definitions stored in the KB:

```
Class ERGraphEDGE in AnswerRepresentation isA EDGE with
        graphicalType
                gT1 : ERobjects;
                gT2 : Relationship
                gT3 : Feature;
                gT4 : Class
end ERGraphEDGE

ERGraphEDGE!gT1 with
        orderValue
                v : 1
end ERGraphEDGE!gT1

ERGraphEDGE!gT2 with
        orderValue
                v : 2
end ERGraphEDGE!gT2

ERGraphEDGE!gT3 with
        orderValue
        v : 3
end ERGraphEDGE!gT3

ERGraphEDGE!gT4 with
        orderValue
                v : 4
end ERGraphEDGE!gT4
```

and corresponding descriptions for a class `ERGraphNODE` (compare section 3.2.5).

The configuration file `.graphbrowserinit` for the graphical browser used in this example contains among others the following lines which fix the relationship between the graphical types of objects (derived from the format definitions in the KB) and their graphical representation:

```
CLASS,black,rectangle,GB_NODE|GB_SOMETHING|GB_TOOLS
ERobjects,gray,rectangle,GB_NODE|GB_SOMETHING|GB_TOOLS
Relationship,none,oval,GB_NODE|GB_SOMETHING|GB_TOOLS
Feature,none,icon,GB_NODE|GB_SOMETHING|GB_TOOLS
```

Shape value `icon` without any specific designed icon leads to the non surrounded label representation of features.

abcd

Appendix 4: CoAUTHOR Tool

A4.1 Conceptual Model of CoAuthoring

The *CoAUTHOR* tool provides a real-time oriented environment for multiple authors who wish to collaborate on the production of hypermedia documents. This section describes the conceptual model serving as a backbone of the tool and its user interface. Section A4.2 illustrates the use of the tool by an example session. There are two dimensions of conceptual models; first, a model of authoring including a document model, and second, a model of group work.

The model of authoring is tri-partite. The top level of *idea processing* determines the issues which have to be covered by the resulting document. This includes the creation, hierarchical refinement, and definition of relations among ideas by semantic-bearing links. At the middle level of *document design* a formal document strucure has to be set up and associated with the conceptual items from the idea processing level. Again, there are relationships - hierarchical and precedence relations - between those structure objects. Finally, at the bottom level of *document generation* ideas get implemented by an appropriate hypermedia chunk, relationships here as typed hypertext links.

This model has been implemented in Telos and is stored and maintained by the *ConceptBase* server. Within the KBMS there is an overall class of objects, called `CoAUTHOR_Object`, with three subclasses according to the three levels of authoring: `Idea`, `Structure`, and `HyperMediaObject`.

Attributes of `Idea` objects are `subIdea` (representing a hierarchical relation between ideas), `relatedTo` (representing user defined relations between ideas), and `refine` (linking a `HyperMediaObject` to the `Idea` object containing a description of the intention of this idea). Attributes of `Structure` objects are `subStructure` (representing a hierarchical relation between structures), `precedeStructure` (representing a precedence relation between structures), and `relatedIdea` (representing the link to the `Idea` object associated to this `Structure` object). `HyperMediaObjects` have attributes `relatedStructure` (representing the link to the `Structure` object associated to this hypermedia chunk) and `hypertextLinks` (representing - at the conceptual side - the hypertext links between those chunks). See Figure A4-1 for a graphical illustration of the model of authoring.

Orthogonal to the model of authoring there is a conceptual model of group work, which is also characterized by a partition into three major phases. Group work starts with the *generation* of individual contributions at each level of the authoring task, such as ideas, structures, and specific hypermedia chunks. At the upper two levels this could be done at different times or within real-time conferences, at the document level it has to be done separately by each author.

Each of these contributed objects and even the relations between them are open for *annotation* by other group members, again within conferences or asynchronously. Annotations may be the attachement of relevance assessments, critical comments, the augmentation by alternatives and so on.

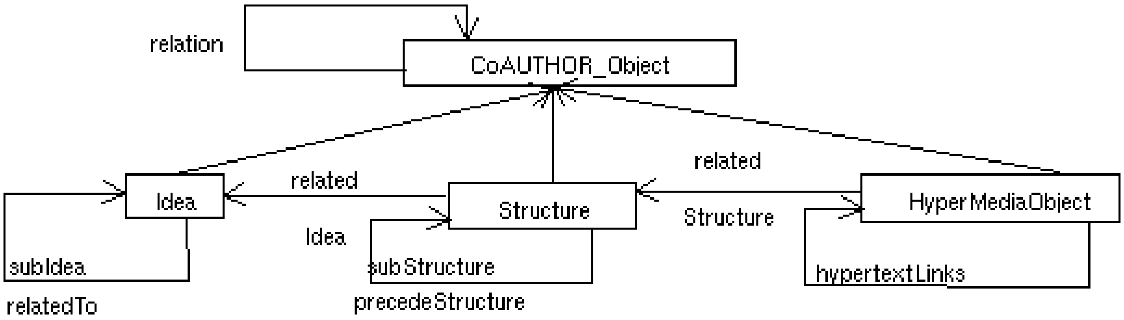


Fig. A4-1: conceptual model of authoring

Finally, the basic building blocks of a document have to be combined to form larger units on each of the three levels. This is essentially a *configuration* task. During this phase incomplete specifications, inconsistencies, and multiple solutions are likely to emerge at different levels of the authoring process. Negotiation-based alignment strategies have to support the resolution of those conflicts. As a result, ideas, structure objects, and concrete hypermedia chunks can either be confirmed, modified or withdrawn, leading to a global readjustment of the original contributions.

Analogous to the model of authoring the model of group work has been implemented in Telos. A central aspect of this model is an overall `Annotation` class which is subclassed by a `Pro`, a `Counter`, and an `Alternative` class according to the *annotation* layer of the group work described above. `Pro` and `Counter` objects represent positive and negative comments attached to objects at every level of the authoring process. For detailed comments, it is possible to attach a hypermedia chunk to a comment object. `Alternative` objects are designated to state common properties of two objects at any level. Each of these objects is related either to one of the specializations of `CoAUTHOR_Object` or any relation between those objects (figure A4-2).

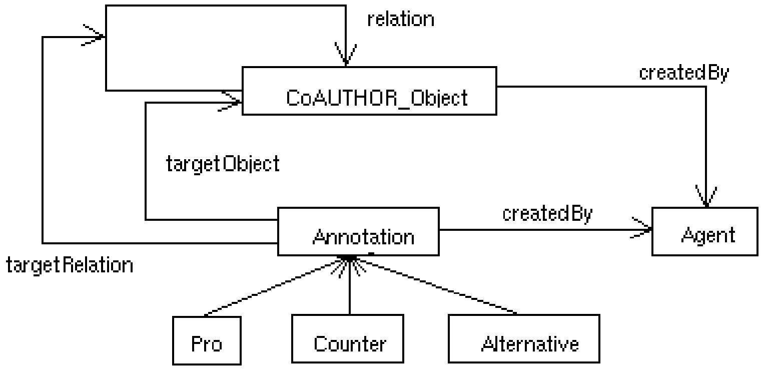


Fig. A4-2: group model of authoring

Another important part of the group model is the concept of agents, which are attached to each of the objects described above to assign a kind of authorship to objects, relations between them, and annotations. We have used a simple single level of agents with equal rights. It is not difficult to introduce a more sophisticated model of agents and actions.

For storing results of conflict resolution processes (like voting etc.) `CoAUTHOR_Objects` got an attribute `status` which can be either IN, OUT or PENDING. A status of IN (OUT) means, that there was a conflict and the group decided, that this object has (not) to be elaborated in the final document. A status of PENDING means, that further negotiations have to take place to solve the conflict. A missing status means, that the object is IN without any negotiation.

Via these authorship links and the annotation links mentioned above the models of authoring and group work have been integrated. This integrated model serves as a formal backbone of the *CoAUTHOR* system (figure A4-3).

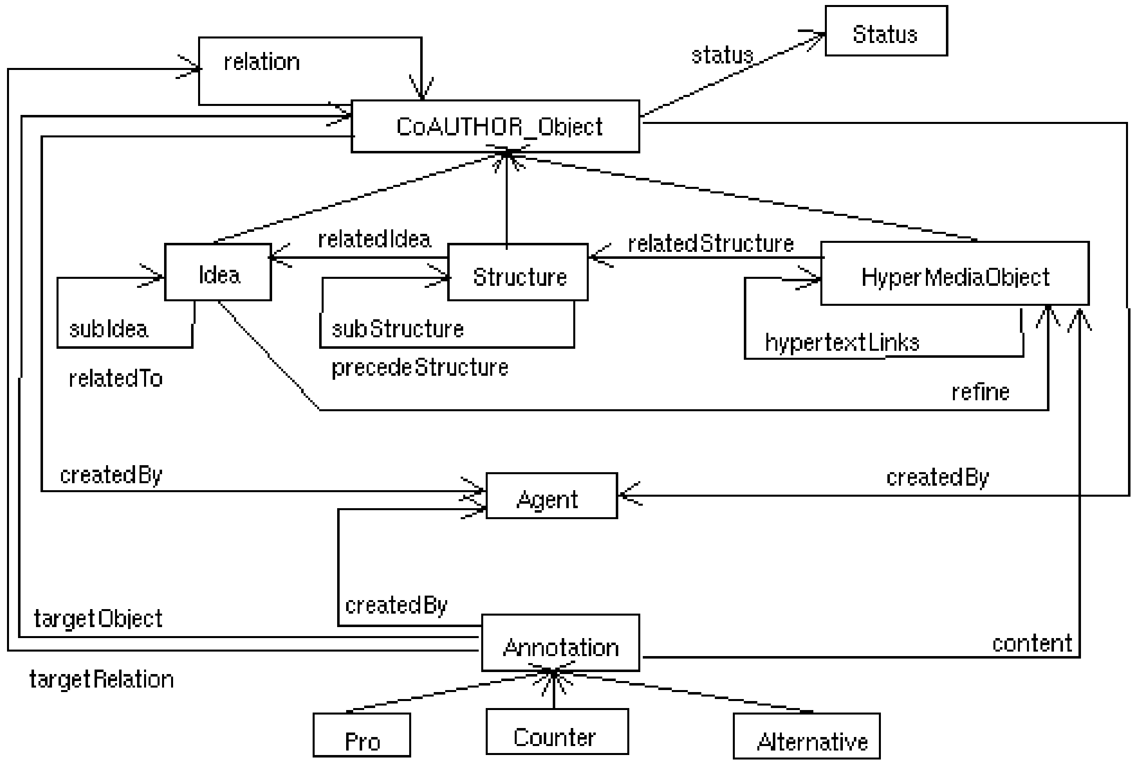


Fig. A4-3: integrated model of co-authoring

A4.2 Features of the CoAUTHOR Tool

The specialized graphical *CoAUTHOR* tool supports the integrated model of co-authoring described in section A4.1. According to the three layers of the authoring model (idea processing, document design, document generation) the tool's window is divided into three major sections.

The IDEAS-section provides menu items for creating new ideas, relating them, and refining ideas by hypermedia chunks. Analogously, the STRUCTURES-section offers menu items for creating new structure objects and relating them. Additionally, structure objects can be related to ideas. The bottom HYPERMEDIA-section provides menu items for introducing documents created with the Andrew document editor into the authoring model and for editing existing documents, editing in this context means retrieving the document from the UNIX file system and displaying it on the screen.

To support the annotation phase of the group model, the tool provides menu items for attaching positive or negative comments to objects (ideas, structures etc.) or relations and for introducing alternatives to existing objects. Each comment consists of a brief description (the label of the comment object when it is displayed within the tool) and possibly some existing document which contains a more elaborated version of the comment.

To support the configuration phase of the group model, the *CoAUTHOR* tool provides a simple real-time conference facility. Several of these tools work on the same knowledge base and every change within one of the tools is passed onto the other tools. To handle parallel changes a simple chalkboard protocol has been implemented: only one of the group members can actually edit the knowledge base, the others can only see the changes made. To get the chalk, in order to edit, one has to ask for it and is then added at the end of a global queue, which will be served sequentially.

A conference is set up by the so-called master. The master has some additional facilities like grabbing the chalk or passing the chalk directly to a specific member.

To detect conflicts due to positive and negative comments attached to the same object or introducing alternatives the tool provides a menu containing the following maintenance operators: the *Show Conflicts* operator returns all those objects and relations which have been assigned positive as well as negative comments; the *Show Alternatives* operator searches for those objects which are linked to an `Alternative` object; the *Show Open Issues* operator returns all those ideas, structure objects, and hypermedia objects whose status is PENDING.

To resolve such conflicts, a voting mechanism helps to aggregate the opinions of the member with respect to a certain object. The evaluation of them might be done according to the strategies proposed by group decision support people. Currently, one can only attach integers within a range between zero and hundred to a certain object. The algebraic average of these integers determines the result of the voting according to the following table:

IN, if the average is greater than 60
PENDING, if the average is between 40 and 60
OUT, if the average is less than 40.

A4.3 Example Session

To illustrate the usage of the *CoAUTHOR* tool an example conference is presented. To run the session the `coauthor` application located in the directory `$CB_HOME/CB_Exe/Examples/COAUTHOR` has to be loaded first[1] . Assume, that a group of authors has to write the documentation of the *ConceptBase* system. They create an overall `Idea` object called `CB_UserManual_IDE`, with several subideas including an idea `TelosEditor_IDE`. Further, a subgroup of three authors has to write the documentation of the *Telos Editor*. First, they elaborate the ideas which should be covered by the final document as subideas of `TelosEditor_IDE` and, based on these ideas, a document structure is created. Each of these activities is done within a real-time conference. Designing the concrete hypermedia chunks, based on the document structure, is done by the authors separately.

The example session starts with setting up a conference. The group leader starts up a *CoAUTHOR* tool and types the name of the starting object, `TelosEditor_IDE`, into the appearing interaction window. The other group members also start up their own *CoAUTHOR* tools specifying a '`$`' as initial input indicating, that their tools should wait for conference set up. The master sets up the conference by choosing the **Conference Set Up** menu item from the Conference panel. This causes a window to come up containing a list of all users currently running a *CoAUTHOR* tool connected to the *ConceptBase* server. The master chooses the users he wants to include into the conference and communicates his initial data to them (figure A4-4).

---

[1] Note, that for convenience, all objects mentioned are already defined in this application.

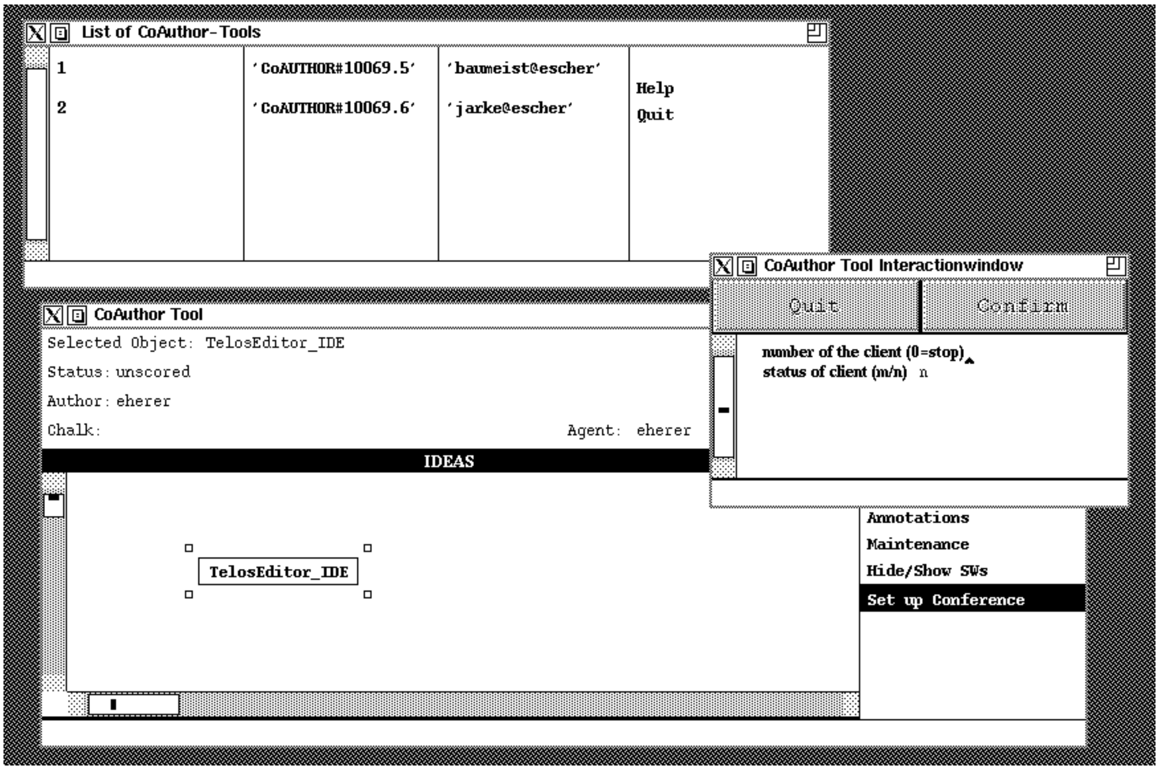


Fig. A4-4: Setting up a conference

The master introduces a subidea of `TelosEditor_IDE` by choosing the **Refine Idea Hierarchy** menu item and specifying the name of the new idea, `EditingFacilities_IDE`. To elaborate the intention he attaches a previously created hypermedia object to his idea (choosing the **Refine Idea** menu item of the Idea panel).[2] The new objects are communicated to the *ConceptBase* server and to the other participants to guarantee, that each participant works on the same data.

Another participant looks at this refinement by choosing the **Get Refining Document** menu item. He wants to make a comment to the `EditingFacilities_IDE` object. Therefore he asks for the chalk (choosing the **Ask Chalk** menu item). The master is informed, that another user wants the chalk and releases it (choosing the **Release Chalk** menu item). After getting the chalk the requestor introduces his negative comment `"Trivial"` (choosing the **Contra Object** menu item). Giving `"none"` as attached document he indicates, that there is no further elaboration. Additionally, he introduces a subidea of `TelosEditor_IDE` called `EditorOperations_IDE`. Each alternation is immediately communicated to all group members and the *ConceptBase* server.

Now, the third group member wants to endorse the `EditingFacilities_IDE` idea and therefore asks for the chalk. After getting it he attaches a positive comment `"Needed"` by choosing **Pro Object** (figure A4-5). Now there are three idea objects, one of them refined and one of them with two comments attached.

The master chooses the **Show Conflict** menu item to look for existing conflicts. Because of the negative and positive comments attched to the `EditingFacilities_IDE` object there is a conflict. The effected object is highlighted.

---

[2] The new idea is displayed as a dark rectangle to indicate, that this idea is refined.

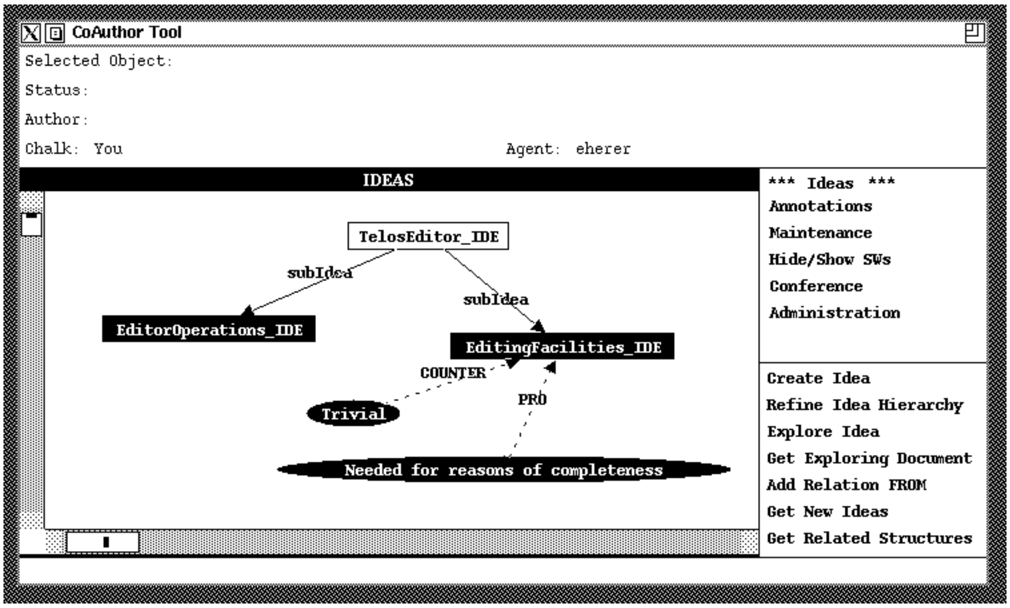


Fig. A4-5: Situation after the idea processing phase

To solve this conflict the master selects the highlighted object and chooses the **Do Vote** menu item. Now every participant is called on to vote (giving an integer between 0 and 100 indicating a weight) on this object. In our example the master and the 'positive' participant give a weight of 100 and 90 respectively, whereas the 'negative' participant gives a weight of 0. Each vote is sent to the master tool where the result (IN) of the voting is computed. The new status of the object is communicated to the *ConceptBase* server as well as to every group member (figure A4-6).

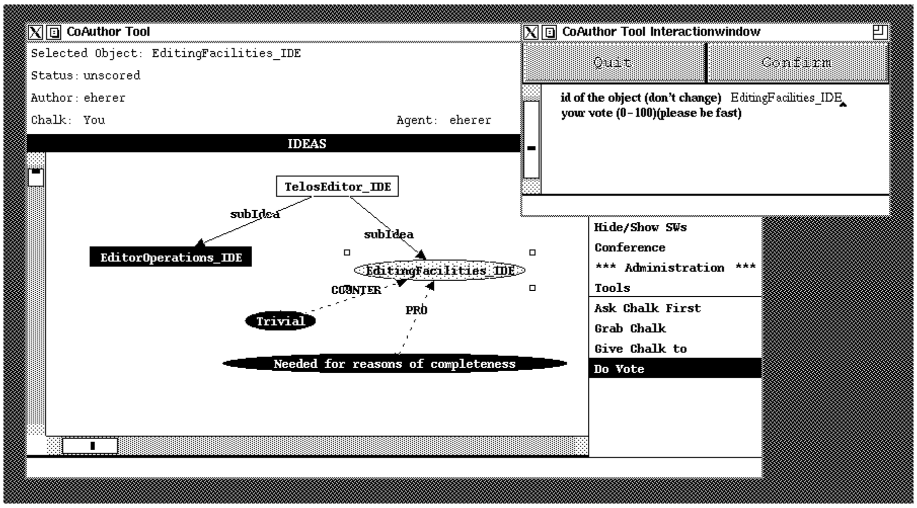


Fig. A4-6: *CoAUTHOR* tool ready for voting

Now the master decides to finish the idea processing phase. He communicates with the group, that they now enter the process of document design. The master first grabs the chalk (choosing the **Grab Chalk** menu item) and then creates a new structure object, let's say `TelosEditor_STR` (choosing the **Create Structure** menu item). Finally, he relates the new structure object to the existing `TelosEditor_IDE` idea object (choosing the **Structure To Idea Mapping** menu item).

In the following the group creates two additional structure objects, `EditingFacilities_STR` and `EditorOperations_STR`, as substructures of `TelosEditor_STR` and relates them to the idea objects `EditingFacilities_IDE` and `EditorOperations_IDE` respectively. Now, the master asks for the chalk and, after getting it, creates a precedence relation from the `EditingFacilities_STR` to the `EditorOperations_STR` object (choosing the **Precede Structure** menu item).

See figure A4-7 for the result of our sample document design process; there are three structure objects, all of them already mapped to idea objects, and there is a precedence relation between two of them (displayed as a thick arrow).

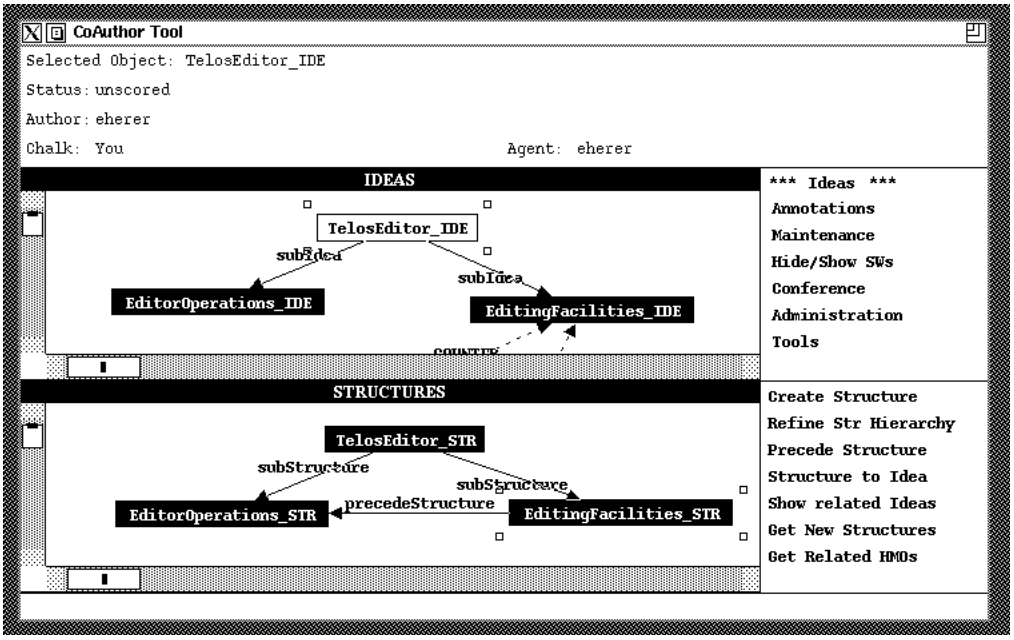


Fig. A4-7: Situation after the document design phase

Assume, that the 'positive' participant agreed to implement the `EditingFacilities_STR` object, the 'negative' one the `EditingOperations_STR` respectively. The conference is closed now (choosing the **Leave Conference** menu item) and both ot them start up a new *CoAUTHOR* tool with `EditingFacility_IDE` and `EditingOperations_IDE` respectively as initial idea object. In the following we concentrate on the 'negative' one to illustrate the steps to be done in order to create a concrete hypermedia object.

He invokes the hypertext editor from the editing panel of the *ConceptBase Tool Bar* and creates a new hypermedia chunk, called `EditorOperations_HMO`. Then he types in the text describing the editor operations and adds a screendump of the editor's window. To introduce the new chunk into the co-authoring process he chooses the **Use Object** menu item and relates it to 'his' structure object, `EditorOperations_STR`.

Simultaneously, the other author (the 'positive' one) created a hypermedia chunk (called `EditorFacilities_HMO`) implementing the `EditorFacilities_STR` structure object and introduced it into the co-authoring process.

In order to configure the group document another conference is set up during which the group decides, that a hypermedia link should be created between `EditorFacilities_HMO` and `EditorOperations_HMO`. This is done within the *Hypertext Editor* by choosing the item **Add Link** and then **Tell & Save** in order to communicate the new link to the *ConceptBase* server. Figure A4-8 shows both new documents (`EditingFacilities_HMO` with the link in it) and the final status of the *CoAUTHOR* tools.

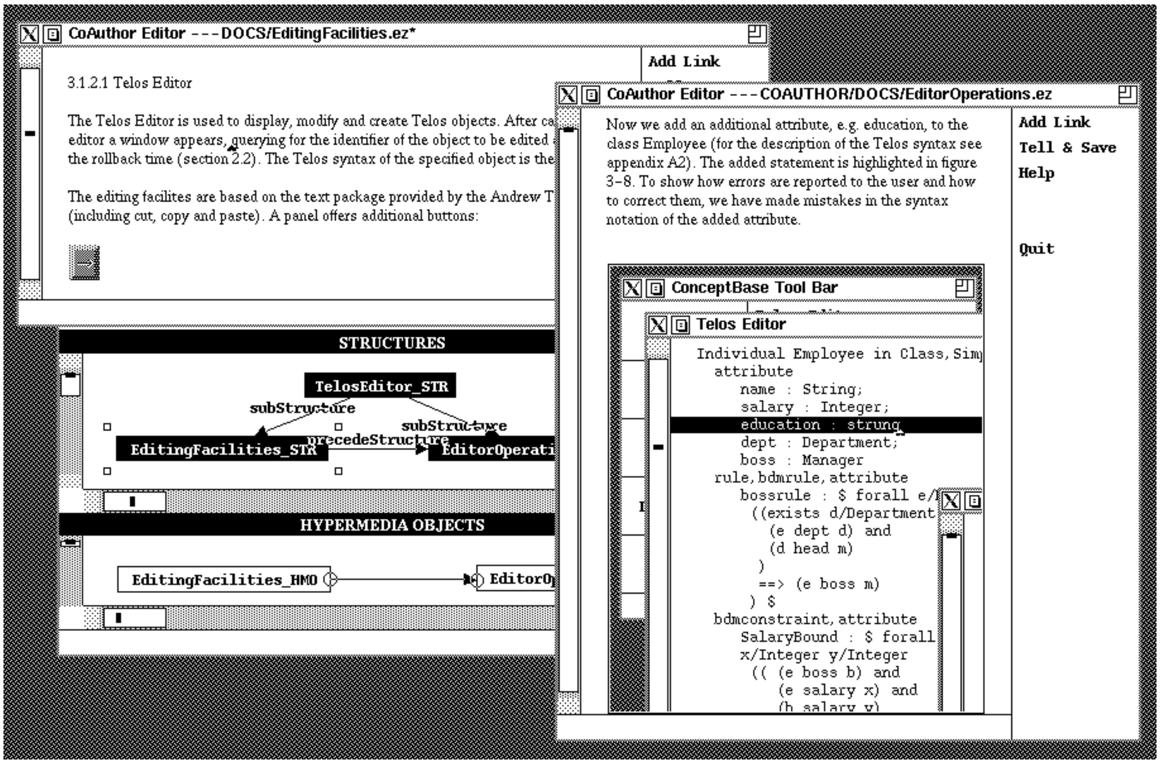


Fig. A4-8: after creating hypermedia objects