TROEPS 1.3a reference manual

(Modified: 21/12/99 16:05, Printed: 21/12/99 16:08)

Sherpa project

INRIA RHÔNE-ALPES

655 avenue de l'Europe, 38330 MONTBONNOT-SAINT MARTIN, France

troeps@inrialpes.fr

Foreword

TROEPS is an object based knowledge representation system. It thus allows to model particular domains under the form of objects. This means that the individual entities of the domain are seen as discrete entities with fields denoting their characteristics and their relationship with other such entities. These objects are also grouped into families of objects called concepts or classes which, in turn, can be related to each others by specialisation or attributive relationship. Each construct of the system has a precise semantics which allows to check the models for internal consistency.

TROEPS knowledge bases can be explored in order to find the objects and their relationships. To that extent, turning a knowledge base into a Web knowledge server, either for browsing, editing or exploiting the model, has been made easy. Moreover, lexicons and annotations enable to document the model straightforwardly.

However, the interpretation and exploitation of the model is left to external programs (generally through the TROEPS API). The system provides basic tools for easing the use of models such as sophisticated field value computation, instance classification and conceptual clustering.

This Reference manual provides an in depth presentation of the TROEPS system. It can be found on-line at `http://co4.inrialpes.fr/docs/troeps-manual.html`.

About TROEPS

TROEPS can be considered as the successor of SHIRKA. It thus shares with SHIRKA the notions of objects, classes, taxonomy, hypertext links and classification facilities. It also adds the possibility of having several taxonomies over the same set of objects, extendibility through abstract data types and constraint satisfaction.

TROEPS is a result of the Sherpa project headed by François Rechenmann from 1988 to 1999. The system stems from the work of the whole Sherpa team and the previous research on SHIRKA and continuous efforts made in order to enhance it. Meanwhile, the overall design of TROEPS has been the early initiative of Olga Mariño, the development of TROEPS has been carried out at various level of involvement by Christophe Alemany, Cécile Capponi, Isabelle Crampé, Pierre Fontanille, Pierre Girard, Jérôme Gensel, Claire Lecourtier, Florence Lemaire, Olivier Schmeltzer, Loïc Tricand De La Goutte, Nina Tayar and Petko Valtchev under the supervision of Jérôme Euzenat.

The actual implemented system is not faithful to the design of Olga Mariño's papers. Some notions (like annotations, lexicon, categorisation) have been added to the system, others (like the semantics of taxonomies) have been changed but the main advantage of the current design is that it works and is implemented. Nevertheless, the implemented system is not the final one. The implementation has been carried out in such a modular way that it will be possible to replace a straightforward implementation by another one (concerning, for example, the memory storage or the basic classification mechanisms) or to add new modules (for instance, clustering facilities or a complete revision mechanism).

The extensions of the system such as the knowledge server are built on top of an API (Application Programming Interface). That API which represents the core of the system is implemented and described in the present manual.

Beware: although other documents, anterior or posterior, can help to understand particular aspects of the system, **this reference manual is the only trustworthy documentation of TROEPS 1.3 made available from december 1999**.

Organisation and notation

As said above, TROEPS is made of a restricted set of basic notions. These notions are described in this manual in a constant frame consisting of:

- an explanation of the notion,
- the description of the graphic interaction involved in the manipulation of the notion (through Web pages),
- the complete specification of the API about that notion.
- the presentation of API manipulation through an example.

Among the particular notations used in this manual, functions of the API are presented in the following format:

(**primitive-name** *argument*$_1$… *argument*$_n$) primitive-type in [library-name]

-> returned-type, *argument*$_1$ argument$_1$-type$_{\text{error-types}}$…, *argument*$_n$ argument$_n$-type$_{\text{error-types}}$

Description of the primitive behaviour.

Type expressions are expressed in a kind of Backus-Naur form. The error names can be found in Appendix 7. Sometimes, the mention "NIY" and a "—" sign after the closing parenthesis signals that the primitive is not available in the current version. There are very few non available primitives.

Interface are displayed through particular figures called Screendumps. They provide a picture of the HTTP browser screen together with the indications of the next screens when clicking on an active element (link or button). Some screens (mainly dialogues) are omitted since they do not carry a lot of information and other ones are reduced through white stripes across them.

Screendump <number>: <Description of the screen function>. <Additional information>

<link>: ..... see Screendump <other number>, p. <pagenumber>

…

<Button>:..... see Screendump <other number>, p. <pagenumber>

…

The examples are given in their TALK form (i.e. not in a C form, however, the use of TALK constructions is limited to the well-known list operations: `cons`, `car`, `cdr`, `list` and `eq`). They constitute the creation of a knowledge base extending an already existing base (`real-estate` available in the examples, which, by opposition, is the subject of the examples given in the text). So, each example session can be played only after having played the former sessions. A primitive is presented in both the section corresponding to its API description and/or in the section corresponding to its arguments. Finally, the format of TALK examples is made of a form to evaluate (preceded by a question mark) and the results of the evaluation (preceded by an equal sign). The result is printed by the TROEPS printer which displays a simple readable form for TROEPS entities. For instance,

```
? (tr-find-concept "house")
= (tr-concept "house")
```

corresponds to the evaluation of `(tr-find-concept "house")` which returns the "house" concept printed as `(tr-concept "house")`. This is useful since the result can be cut and pasted in order to provide a reference to the object.

Hereafter, the "designer" will not be considered as different from the "user": a designer has, *a priori*, no additional facilities to use a knowledge base. So, "user" will be the general term used, and "designer" will be reserved for discussions on strict design problems, if any.

Plan

The manual is made of three main chapters. The first one, *Data model*, describes the static model used in TROEPS in order to represent knowledge in term of objects. The second one, *Computing field values*, is dedicated to the dynamic aspect of inferring field values. The third one, *Classification principles*, concerns a very important aspect of TROEPS: the possibility to find the most suited class for an object with regard to the overall knowledge base. A fourth chapter concerns the setting of a *Knowledge server* taking advantage of the World wide web.

A set of appendices is added to these four main chapters. They concern:

1) how to get and install TROEPS;
2) the textual format (how to describe knowledge bases in files);
3) an example of knowledge base (TROEPS itself);
4) the TROEPS document type description
5) how to add data types into TROEPS;
6) how to launch and catch errors;
7) the list of error messages;
8) the list of reserved annotation labels;
9) a glossary;
10) the index of functions.

The novice reader is invited to read the manual through the three main sections; it is possible to reproduce the examples from the first to the last. The installer and/or newcomer should consult the Appendix 1 for a quick start. The programmer will access directly the manual from the function index at the end of the manual or by browsing the on-line manual through a WWW client (see Appendix 1).

Montbonnot, le 20 décembre 1999

Table of contents

FOREWORD **3**

ABOUT TROEPS 3
ORGANISATION AND NOTATION 4
PLAN 5

TABLE OF CONTENTS **6**

DATA MODEL **8**

KNOWLEDGE BASE 12
CONCEPTS 19
FIELD DEFINITION: CONCEPTUAL ASPECTS 27
VIEWPOINTS 33
CLASS DEFINITION 38
CLASS FIELD DEFINITION: DATA TYPING ASPECTS/UNARY CONSTRAINTS 46
FILTERS 55
BRIDGES 59
OBJECTS 65
ANNOTATIONS 74
LEXICON 76

COMPUTING FIELD VALUES **84**

IMPLICIT VERSUS HYPOTHETICAL COMPUTATION 84
HYPOTHETICAL COMPUTATION, INHERITANCE BY DEFAULT AND ORDERING 85
DEFAULT VALUE 88
PATHS AND SLOT VALUE PASSING 89
COMPUTATION BY FILTER 90
TASK INVOCATION 92
CONSTRAINTS 94

CLASSIFICATION **97**

INSTANCE CLASSIFICATION 97
CATEGORISATION 100
CLUSTERING 100
BRIDGE INFERENCE 107
COMPLETE EXAMPLE 109

KNOWLEDGE SERVERS **113**

SETTING UP YOUR OWN SERVER 113
EDITING KNOWLEDGE BASES THROUGH THE SERVER 115
CUSTOMISING THE APPEARANCE 115
CONCURRENT EDITING WITH CO4 119

APPENDIX 1. HOW TO GET, INSTALL AND RUN TROEPS 121

APPENDIX 2. TEXTUAL FORMAT 125

APPENDIX 3. TROEPS IN TROEPS TEXTUAL FORMAT 132

APPENDIX 4. TROEPS DTD 135

APPENDIX 5. ADDING DATA TYPES TO TROEPS 137

APPENDIX 6. ERROR SYSTEM 142

APPENDIX 7. ERROR MESSAGES 146

APPENDIX 8. RESERVED ANNOTATION LABELS 148

APPENDIX 9. GLOSSARY 149

APPENDIX 10. FUNCTION INDEX 155

Data model

Knowledge representation aims at representing a particular domain and reasoning about it. This is what TROEPS is about. A knowledge base is the repository for the pieces of knowledge on a domain. It can also use other knowledge bases and resources. It is made of four kinds of knowledge:

- *Descriptive* knowledge is represented in an object-based knowledge representation system. This enables the representation of classes of objects (e.g. house), subclasses (e.g. palace) and the identification of an object as belonging to such a class. It provides constraint satisfaction and typing mechanisms for the objects.
- *Methodological* knowledge specifies the ways to select and link up methods for a given task. It is represented through a task management system able to integrate, represent, process and monitor computer programs.
- *Behavioural* knowledge concerns the modelling of dynamic phenomena, such as the dynamics of contract negotiation between buyers and sellers, through a qualitative modelling system.
- *Textual annotations* on the various objects and tasks involved are achieved through a hypertext system which connects hypertext nodes with the components of the descriptive and methodological knowledge. It allows browsing among texts, objects and tasks. Here textual annotations are not part of the TROEPS model: in TROEPS, the language for expressing them is the HTML standard.

In the current state of TROEPS, only the first and last part are available and described in the manual.

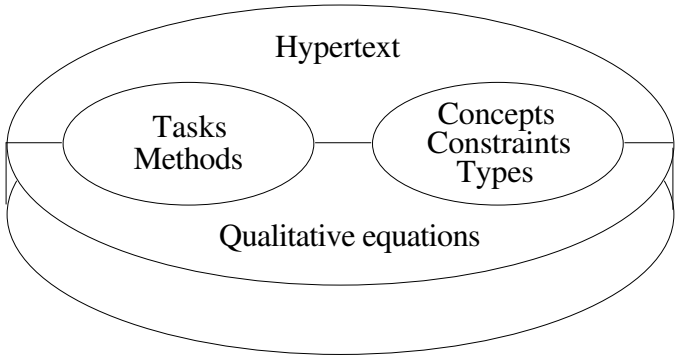


Figure 1: The four parts of a knowledge base.

TROEPS is an object-based knowledge representation model. This means that individuals are represented as objects. These objects are partitioned in concepts (an object is instance of one and only one concept). An object is thus an instance of a concept; it is attributed a set of fields which are described in the concept. A field expresses either a property, a "part-of" relation link, or a relational link. Fields thus allow to express the relationship between objects (and concepts). The different perspectives one can have on an object are represented through viewpoints: a viewpoint determines first the set of fields which are relevant (or visible) and a hierarchy of classes in which the instances of the concept can be classified under that viewpoint. Classes introduce constraints on the field values of objects which are member of them. They are related through the specialisation relation which determines progressive subsets

of the set of the instances of the concept (see Figure 2). This progression is supported by the progression of the constraints added from specialisations to the classes they specialise. In each viewpoint, an object is attached to only one class, but is a member of all the classes of which this class is a specialisation. Class taxonomies are the support of powerful classification mechanisms described in the third section. Viewpoints allow one to focus only on the relevant aspects of objects while hiding those which are not. A bridge can be thrown from a set of classes (called the source) to another class (called the destination) of different viewpoints in a concept in order to signify that the instances which are member of all the source classes are also member of the destination class.

For instance, if the domain to be modelled is that of a real estate agency, three viewpoints can be put forth for the `house` concept. These viewpoints cover the topics of `localisation`, `size` and `price` on the `house` concept. The `localisation` viewpoint distinguishes `center` apartments, `suburban` houses and `other-area` among which `residential` and `industrial` places. The `size` viewpoint differentiates flats by the number of their rooms (usually normalised as `studio`, `F1`, `F2`, `F3`... `palace`). Lastly, from the `price` viewpoint there are three rent categories: `low`, `average` and `high` price.

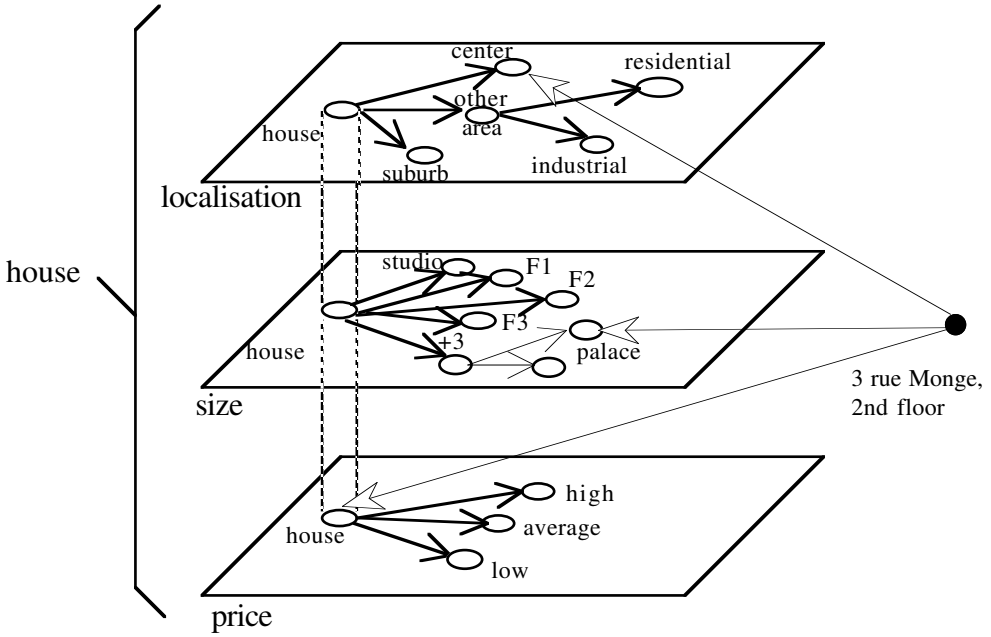
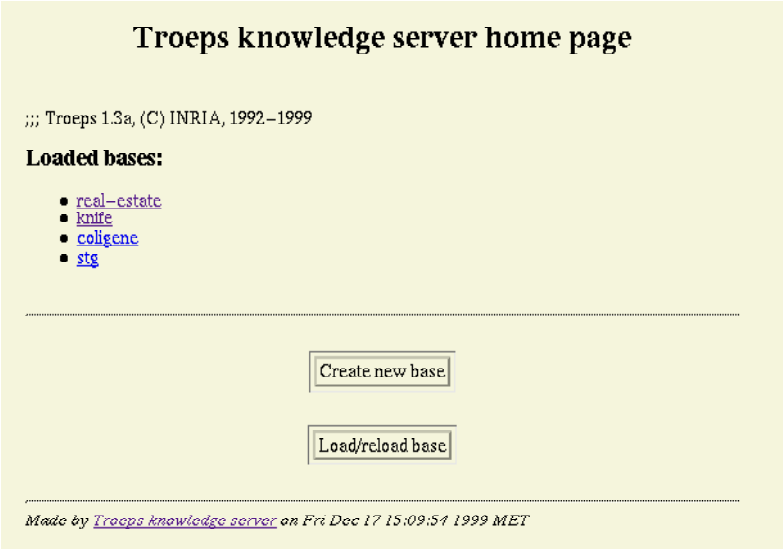


Figure 2: The `house` concept in the `real-estate` knowledge base.

The present chapter explores the different entities that are to be understood in order to describe objects in a knowledge base. It does this by going from the more general notion (knowledge base) to the more specific (object). However the reader should keep in mind the headlines drawn above in order to get familiarised with each notion presented here. A glossary, in Appendix 9, can help to remind some specific notions.

Interface



Screendump 1: TROEPS server home page.

<base>:.....see Screendump 2, p. 14

API

(tr-init) function in [libtrp]

-> boolean

Initialises the TROEPS system by defining the primitive notions which are described hereafter. Calling this function is compulsory before any use of TROEPS. It can be used in order to re-initialise the whole system: then, any user-defined notion or loaded knowledge base is discarded from the environment.

(tr-version) function in [libtrp]

-> (integer*)

Prints the version number and returns a list of integers denoting the current version of the system.

Examples

```
$ italk
;;; ILOG Talk version 3.3 [sparc_5_gnu]
;;; Copyright (c) 1989-1997 by ILOG
;;; Write to talk-support@ilog.com for help
;;; Image saved on Fri Jul  4 19:26:59 1997
;;; Connecting to Emacs server...connected.
;;; Loading workspace `talk': Ilog Talk kernel workspace
;;; Loading workspace `trp': Troeps development workspace
;;; Loading workspace `trt': Troeps taxonomy building tool workspace
;;; Loading workspace `htrp': Troeps knowledge server development workspace
;;; Loading workspace `divers': executables and utility files workspace
;;; Welcome to ILOG Talk.
;;; Type (exit) to leave Talk.
? (load-program-unit 'libtrl)
;;; Loading library `libtrl': Troeps textual interface
;;; Loading run-time environment of `libtrl': (libtrp)
;;; Loading library `libtrp': The core Troeps library
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/lib/libtrp.so"
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/lib/libtrl.so"
= libtrl
? (load-program-unit 'trtdum)
;;; Loading module `trtdum': Troeps dummy basic type descriptions
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/o/trtdum.o"
```

```
= trtdum
? (tr-version)
;;; Troeps 1.3a, (C) INRIA, 1992-1999
= (1 3 . 1)
? (add-handler (function troeps-handler))
= #<handler-function troeps-handler #p0019ad70>
? (tr-init)
= t
? (exit)
$
```

Knowledge base

A *knowledge base* groups a set of descriptions concerning the entities occurring in a particular domain. It thus contains a set of concepts (together with their viewpoints, fields, classes, objects and bridges) and a set of knowledge base viewpoints (see below). However, in order to be linked with other domain descriptions, the knowledge base contains reference to a set of other components:

- the name of the file in which the knowledge base is stored;
- the other knowledge bases it imports (this is useful, for instance if the concepts in the base refer to other concepts);
- the set of abstract data types (see Appendix 5) used in the base;
- the set of object files for the knowledge base (for efficiency purpose, they will be disconnected from the knowledge base scheme, thus enabling to be connected to a secondary storage manager);
- the set of TALK modules to be included in the knowledge base (since some programs for manipulating the knowledge could be useful);

For the time being, a knowledge base can be created and manipulated. It can also be set as current so that each newly created concept will be considered as belonging to that base and will be saved with the base. It can also be loaded from and saved on files of various formats which are explained in Appendix 2.

Concerning data types, the knowledge base loader is not able to load type descriptions thus, they must be loaded independently of the knowledge bases. TROEPS is currently provided with a type module (`trtdum`) which declares base types (see Appendix 3).

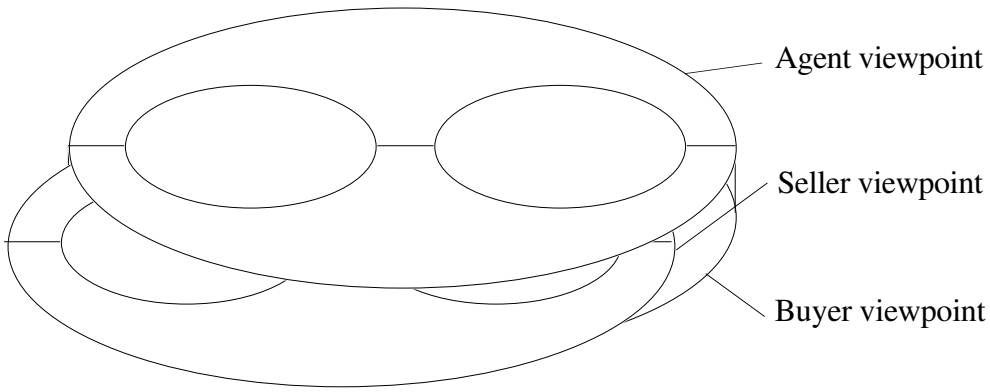


Figure 3: The `real-estate` knowledge base can be used by several people. Then, it can be seen from several viewpoints: `agent`, `seller`, `buyer` (for instance).

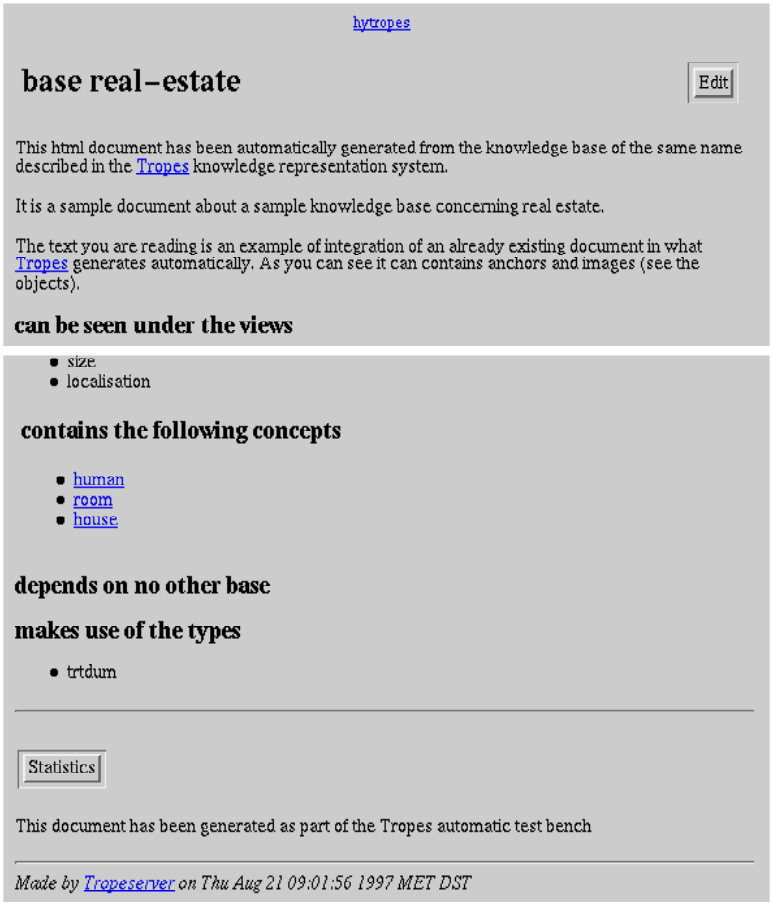
Knowledge base viewpoints

A knowledge base aims at modelling and structuring knowledge. This knowledge may come from different (sub-)domains on one hand, and the users may, on the other hand, consider their own set of knowledge, of properties and relations between pieces of knowledge. So, knowledge has to be organised in such a way that viewpoints may be described. TROEPS includes the *viewpoint* notion. Viewpoints have been designed in order to allow specialists in different domains to express their knowledge independently of one another. A viewpoint represents a specific point of view on the knowledge included in the knowledge base.

Viewpoints are described at the upper level in the knowledge base and can then be viewpoints on the concepts described in the knowledge base.

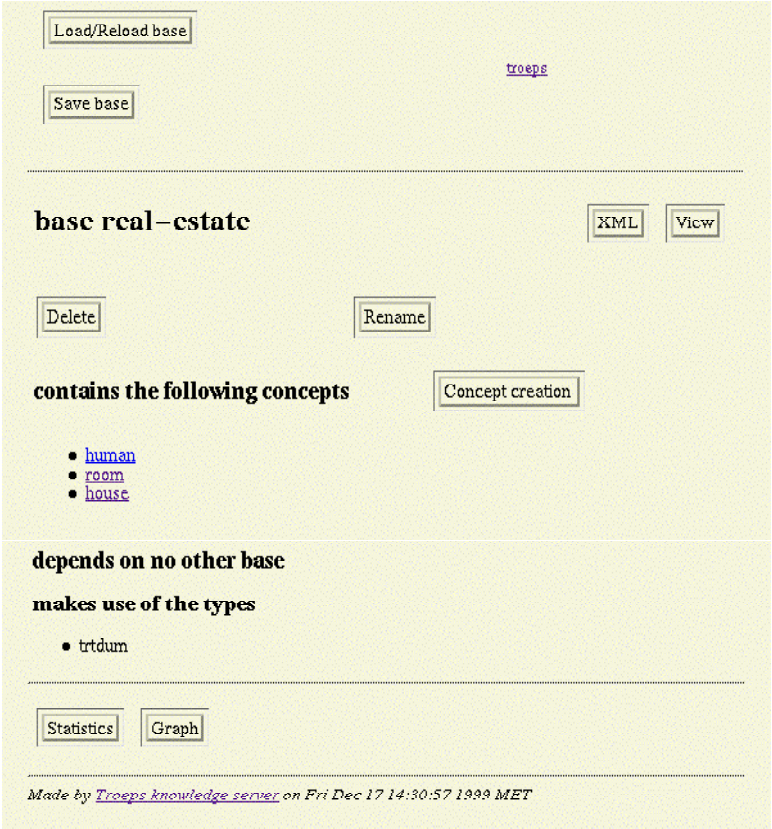
These global viewpoints on the knowledge base do not have special prerogatives in the current implementation (they are presented for completeness purposes). They must be distinguished from the concept viewpoints which will be presented later on.

Interface



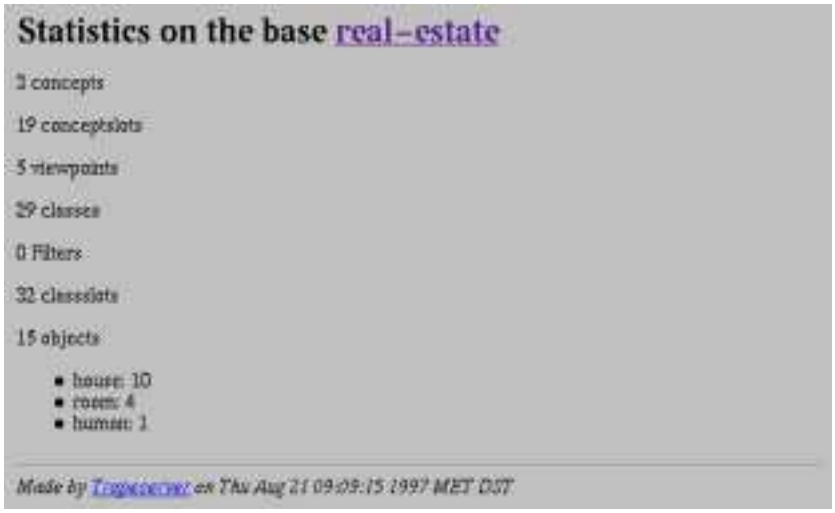
Screendump 2: Knowledge base display.

Statistics .....see Screendump 4, p. 14
Edit: .....see Screendump 3, p. 14
<view>: .....see Screendump 5, p. 22
<base>: .....see Screendump 2



Screendump 3: Knowledge base edition.

View.....see Screendump 2, p. 14
Save:
Concept creation: .....see Screendump 7, p. 24
Statistics: .....see Screendump 4, p. 14
<concept>:.....see Screendump 6, p. 23
<base>:.....see Screendump 3



Screendump 4: Knowledge base statistics.

API

(**tr-base-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a knowledge base, `()` otherwise.

(**tr-create-base** *basename optional: filename*) function in [libtrp]

-> base, *basename* $\text{string}_{\text{nas,adb}}$, *filename* $\text{filename}_{\text{fnf}}$

Creates and returns a new knowledge base of name *basename*. It will therefore be known of the system until deletion or initialisation.

(**tr-find-base** *basename*) function in [libtrp]

-> base, *basename* $\text{string}_{\text{nas}}$

Retrieves and returns the knowledge base of name *basename*. When no such base is known, returns `()`.

(**tr-base** *basename*) macro in [libtrp]

-> base, *basename* $\text{string}_{\text{nas}}$

Retrieves and returns the knowledge base of name *basename*. When no such base is known, creates and returns it.

(**tr-delete-base** *base*) function in [libtrp]

-> boolean, *base* $\text{base}_{\text{nab, neo}}$

Deletes the knowledge base *base* from the environment. This can only happen if the knowledge base is as empty as at its creation. Afterwards, the knowledge base does not exist anymore in the environment.

(**tr-erase-base** *base*) function in [libtrp]

-> boolean, *base* $\text{base}_{\text{nab, neo}}$

Deletes the knowledge base *base* and its content from the environment. Afterwards, neither the knowledge base nor the concepts and objects refered by it exist anymore in the environment.

(**tr-base-name** *base*) function in [libtrp]

-> string, *base* base_{nab}

Returns the name of the knowledge base *base*.

(**tr-base-rename** *base name*) function in [libtrp]

-> string, *base* base_{nab}, *name* $\text{string}_{\text{nas, adb}}$

Changes the name of the knowledge base *base* for the string *name*.

(**tr-base-filename** *base*) setfable function in [libtrp]

-> filename, *base* base_{nab}

Returns the filename of the knowledge base *base*. Can be used for changing that filename.

(**tr-add-viewpoint** *base viewpointnam*e) function in [libtrp]

-> boolean, *base* base_{nab}, *viewpointname* $\text{string}_{\text{nas}}$

Adds the viewpoint name *viewpointname* to the set of viewpoints of the base *base*.

(**tr-rem-viewpoint** *base viewpointname*) function in [libtrp]

-> boolean, *base* base_{nab}, *viewpointname* $\text{string}_{\text{nas,nib}}$

Removes the viewpoint name *viewpointname* from the set of viewpoints of the base *base*.

(**tr-base-viewpoints** *base*) function in [libtrp]

-> { string* }, *base* base_{nab}

Returns the set of viewpoint names of the base *base*.

(**tr-add-concept** *base concept*) function in [libtrp]

-> boolean, *base* base_{nab}, *concept* $\text{concept}_{\text{nak}}$

Adds the concept *concept* to the set of concepts of the base *base*.

(**tr-rem-concept** *base concept*) function in [libtrp]

-> boolean, *base* base_{nab}, *concept* $\text{concept}_{\text{nak,nib}}$

Removes the concept *concept* from the set of concepts of the base *base*.

(**tr-base-concepts** *base*) function in [libtrp]

-> { concept* }, *base* base_{nab}

Returns the set of concepts of the base *base*.

(**tr-base-sorted-concepts** *base optional: sort-function*) function in [libtrp]

-> { concept* }, *base* base_{nab}, *sort-function* function

Returns the set of concepts of the base *base* in their creation order or the order of the base description file by default. If the *sort-function* is provided, the concepts are sorted according to it.

(**tr-add-import** *base base2*) function in [libtrp]

-> boolean, *base* base_{nab}, *base2* base_{nak}

Adds the base *base2* to the set of imported bases of the base *base*.

(**tr-rem-import** *base base2*) function in [libtrp]

-> boolean, *base* base_{nab}, *base2* $\text{base}_{\text{nak,nib}}$

Removes the base *base* from the set of imported bases of the base *base*.

(**tr-base-imports** *base*) function in [libtrp]

-> { base* }, *base* base_{nab}

Returns the set of imported bases of the base *base*.

(**tr-add-type** *base type*) function in [libtrp]

-> boolean, *base* base_{nab}, *type* type_{nak}

Adds the type *type* to the set of types of the base *base*.

(**tr-rem-type** *base type*) function in [libtrp]

-> boolean, *base* base_{nab}, *type* $\text{type}_{\text{nak,nib}}$

Removes the type *type* from the set of types of the base *base*.

(**tr-base-types** *base*) function in [libtrp]

-> { type* }, *base* base_{nab}

Returns the set of types of the base *base*.

(**tr-add-instance-file** *base filename*) function in [libtrp]

-> boolean, *base* base_{nab}, *filename* $\text{string}_{\text{nas}}$

Adds the instance file *filename* to the set of instance files of the base *base*.

(**tr-rem-instance-file** *base filename*) function in [libtrp]

-> boolean, *base* base_{nab}, *filename* $\text{string}_{\text{nas,nib}}$

Removes the instance file *filename* from the set of instance files of the base *base*.

(**tr-base-instance-files** *base*) function in [libtrp]

-> { string* }, *base* base_{nab}

Returns the set of instance files of the base *base*.

(**tr-add-lexicon** *base filename*) function in [libtrp]

-> boolean, *base* base_{nab}, *filename* $\text{string}_{\text{nas}}$

Adds the lexicon file *filename* to the set of lexicons of the base *base*.

(**tr-rem-lexicon** *base filename*) function in [libtrp]

-> boolean, *base* base_{nab}, *filename* $\text{string}_{\text{nas,nib}}$

Removes the lexicon file *filename* from the set of lexicons of the base *base*.

(**tr-base-lexicons** *base*) function in [libtrp]

-> { string* }, *base* base_{nab}

Returns the set of lexicons of the base *base*.

(**tr-add-module** *base modulename*) function in [libtrp]

-> boolean, *base* base_{nab}, *modulename* $\text{string}_{\text{nas}}$

Adds the module name *modulename* to the set of TALK modules of the base *base*.

(**tr-rem-module** *base modulename*) function in [libtrp]

-> boolean, *base* base_{nab}, *modulename* $\text{string}_{\text{nas,nib}}$

Removes the module name *modulename* from the set of TALK modules of the base *base*.

(**tr-base-modules** *base*) function in [libtrp]

-> { modulename* }, *base* base_{nab}

Returns the set of TALK modules of the base *base*.

(**tr-current-base**) setfable function in [libtrp]

-> base, *base* base_{nab}

Returns the current knowledge base. Can be modified by the form:

```
(setf (tr-current-base) base).
```

(**tr-print-report** *base*) function in [libtrp]

-> (), *base* base_{nab}

Prints a statistical report on the number of entities in the base *base*.

Examples

```
? (setf (tr-current-base) (tr-create-base "my-real-estate" #f"my-real-estate.bdf"))
= (tr-base "my-real-estate")
? (tr-base-p ())
= ()
? (tr-base-p (tr-current-base))
= (tr-base "my-real-estate")
? (eq (tr-current-base) (tr-find-base "my-real-estate"))
= t
? (tr-base "my-real-estate")
= (tr-base "my-real-estate")
? (tr-delete-base (tr-base "my-real-estate"))
= t
? (tr-load-base #f"examples/real-estate.bdf")
= t
? (tr-base-name (tr-current-base))
= "real-estate"
? (tr-base-viewpoints (tr-current-base))
= ("localisation" "size" "price" "function" "test")
? (tr-base-concepts (tr-current-base))
= ((tr-concept "house") (tr-concept "room") (tr-concept "human"))
? (tr-base-sorted-concepts (tr-current-base))
```

```
= ((tr-concept "human") (tr-concept "room") (tr-concept "house"))
? (tr-print-report (tr-current-base))
Statistics on the base: real-estate

external types:     0
concepts:     3
conceptslots:    20
conceptviews:    5
classes:    29
filters:    0
classslots:    34
objects:    10
   house:         9
   room:       0
   human:         1
= ()
? (tr-base-filename (tr-current-base))
= #f"examples/real-estate.bdf"
? (tr-erase-base (tr-current-base))
= t
? (tr-load-base #f"examples/real-estate.bdf")
= t
? (setf (tr-current-base)
    (tr-create-base "my-real-estate" #f"my-real-estate.bdf"))
= (tr-base "my-real-estate")
? (tr-add-import (tr-current-base)
          (tr-find-base "real-estate"))
= ((tr-base "real-estate"))
? (tr-base-imports (tr-current-base))
= ((tr-base "real-estate"))
? (tr-base-imports (tr-find-base "real-estate"))
= ()
```

Concepts

Partition of a knowledge base

A knowledge base may distinguish between different and disjoint sets of objects. Let us consider, for instance, the real estate base; it includes families as different as `human`, `house` and `room`. In TROEPS, such a family is called a *concept*. In a knowledge base, concepts define a partition of the universe: any individual object is instance of one and only one concept.

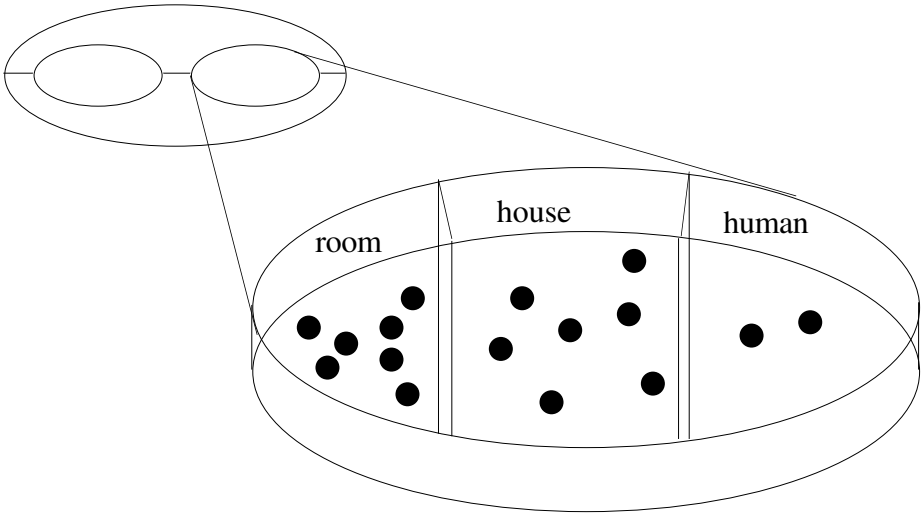


Figure 4: The partition of the `real-estate` knowledge base into several concepts: `room`, `house` and `human`.

Structure of concept instances

Individuals, in a concept, are described by their fields; the same fields and only these fields are used to describe the instances of a concept. For instance, the `house` concept defines the field `address`, `surface`, `floor`, etc. So, the concept is the working space for field names: they are known of the whole concept and, thus, the occurrence of the same field name in the same concept refer to the same field. For instance, the field `age` in the `house` concept represents the age of the building in which the apartment is. But, on another hand, two different fields in two different concepts can have the same name (for instance, the `human` concept can have an `age` field too).

These fields are defined at the concept level. The mandatory field definition describes the whole information valid for any occurrence of it in the concept. It includes:

- the general field type,
- its nature (property, component or relation),
- the constructor, to distinguish sets, lists or atomic values,
- and the tree of methods available to calculate the field value.

These notions are detailed in the “concept field” section.

Identity and integrity

The objects are identified within a concept through a key. This means that, in each concept, a set of fields (called the key) is distinguished. Any instance of the concept is identified by the value of these fields. For instance, the identification of a person can be done by the “social security number” or her/his full name and birthday; an apartment is identified in a

unique manner by the set of fields: `{address, floor, number}`; a "real estate agent" can be defined by its name and the name of its "subsidiary". So, there do not exist two instances with the same values for all the key-fields. Not only the key is unique (which warrants identity) but it is invariant (which warrants integrity): one cannot change what makes the identity of an object. All these properties are ensured by the TROEPS system. Another constraint on the key is that it must be visible under any viewpoint in order to identify the object.

Furthermore, TROEPS allows to handle incomplete instances, but the values of key fields is the minimal information required to create an object and manipulate it. In such a way, the system considers this object as a member of a concept in the knowledge base, so it may identify it as different from other instances.

TROEPS does not handle multiple keys on objects: for instance, an estate agent, usually retrieves the houses from their price and number of rooms rather than from their location. It could be convenient to address the same set of objects by both keys, but TROEPS does not offer it (however, this can be easily added if necessary; but multiple keys would introduce unavoidable functional dependencies which would have to be taken into account by the implementation).

Thus, minimal information for an instance is the name of the concept plus the set of valued fields of the key.

Keys and cycles

The dependency graph between keys (i.e. the graph which links a concept to the concepts which are the type of its key fields) cannot contain cycles. This is motivated by both computational and semantic considerations. For the latter, if it is required for an object to have a defined key in order to exist, then this key must refer to an existing object which does not depend upon the given object. Thus, circularities must be prohibited at the level of the object keys. One could imagine circularities at the concept level which do not imply circularities for the objects. For instance, one can give as the key of the `human` concept the fields `father`, `mother` and `surname` with `father` and `mother` being `human`. This makes sense but can only be interpreted upon an infinite domain of human beings. TROEPS prohibits such kind of non-circular key reference at the instance level which leads to circular reference at the concept level, because it is harmful for the user. As a matter of fact, this raises problems because for creating your first human you must provide his (or her) parents which are also humans. Tackling interpretation over infinite domains of objects could be the subject of a further version of TROEPS introducing the lazy instanciation of keys.

As a consequence of non circularity through keys, concepts can be partially ordered and this order must be respected by the knowledge base file format (see Appendix 2).

In this first version of TROEPS, it is also not possible to have circularities through any kind of slots. This must change soon, but for the time being it is possible to use links (see conceptslot) instead of first class concept slots.

Reference

Concepts are not totally independent, however. Even if concepts are different, they are related in some manner in the universe to be modelled. In TROEPS, this interaction is modelled through the fields and components of the concept instances. For instance, in the `house` concept, the field `inhabitant` has its values in the concept `human`.

In this first version of TROEPS, it is not possible to refer to a previously undefined concept in a concept. This will be changed soon.

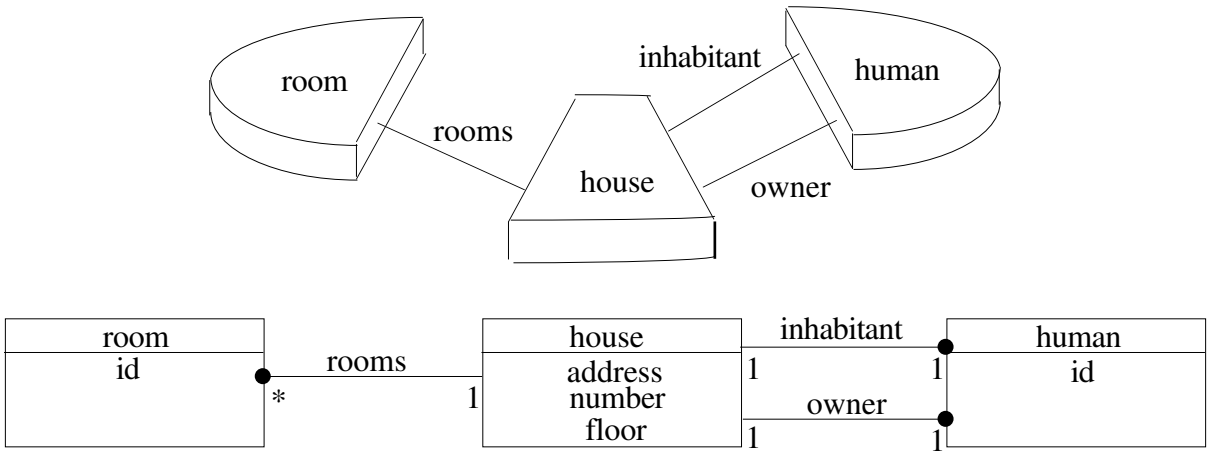


Figure 5: The fields `inhabitant` and `owner` (resp. `rooms`) in the `house` concept has instances of the concept `human` (resp. `room`) as values (upper drawing). Also depicted in an entity relationship diagram (lower one).

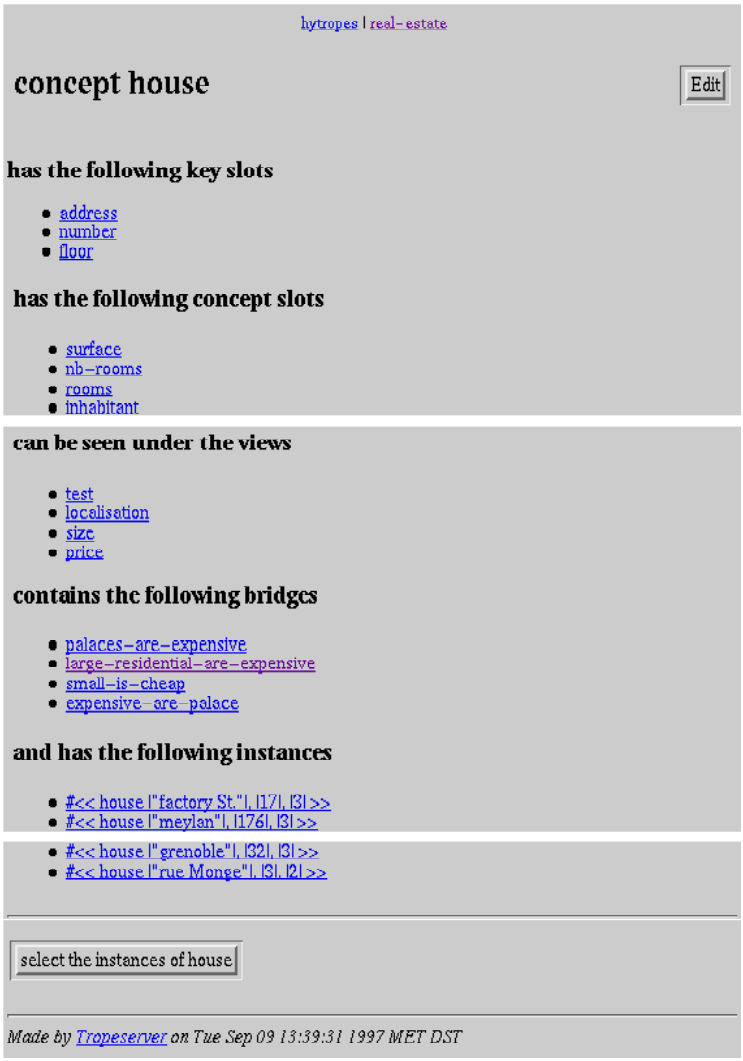
The nature of the reference is modelled through the nature descriptor of the concept field.

Visibility versus typing

The restrictions introduced by the use of viewpoints are manifold. One can distinguish between the hiding of properties and definition of properties. A viewpoint only introduces the hiding of properties: when some fields are not reachable under a particular viewpoint, this does not mean that objects do not have these fields but that they are not visible from the viewpoint. However, objects do have these fields. As a consequence, even if the fields are not visible, they can still introduce some constraints on the object (imagine a magnet, you cannot see the polarity but there are positions in which it cannot remain). This could lead to some surprise for the programmer: the hidden parts of an object still play their role in the shadow. This is mainly relevant for concepts in which there are a lot of bridges and inter-field constraints.

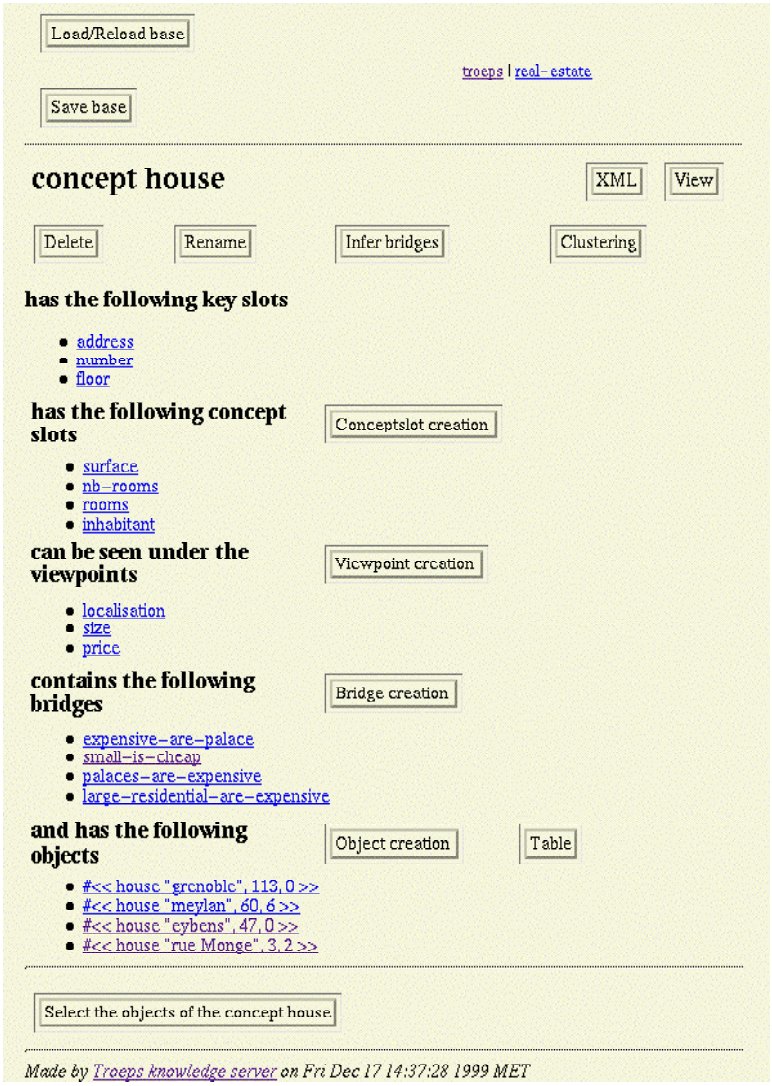
Also, this is related with the definition of field types. When it is said that the type descriptor of a class field can have a set of classes (of different viewpoints) as values, this has to be interpreted as a constraint but not as a visibility information: the constraint is that the value of the field (in order for the object to be in the class) must be member of each of the classes. It does not mean that the value is only visible under the corresponding viewpoints.

Interface



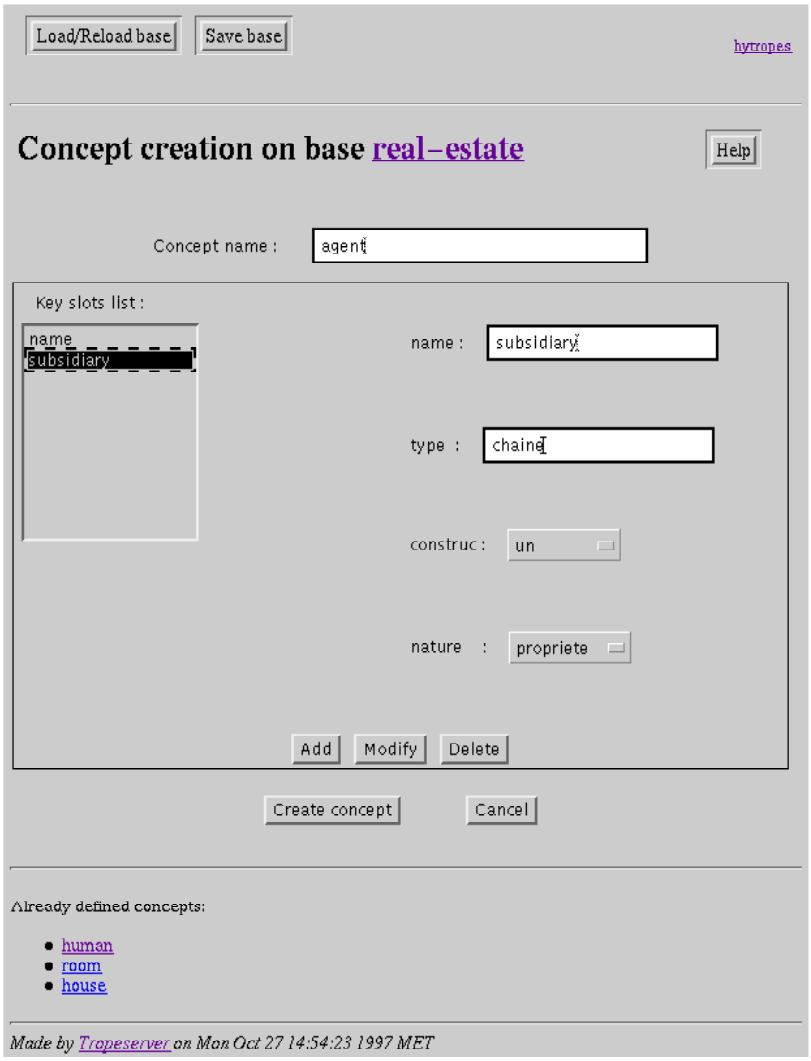
Screendump 5: Concept display.

Edit: ..... see Screendump 6, p. 23
<key slot>: ..... see Screendump 8, p. 29
<slot>: ..... see Screendump 8, p. 29
<view>: ..... see Screendump 11, p. 34
<bridge>: ..... see Screendump 28, p. 61
<instance>: ..... see Screendump 31, p. 66
Select objects: ..... see Screendump 26, p. 56



Screendump 6: Concept edition.

View:.....see Screendump 5, p. 22
Clustering:..... see Screendump 48, p. 104
Bridge inference:..... see Screendump 53, p. 108
<key slot>:.....see Screendump 9, p. 30
Conceptslot creation: ..... see Screendump 10, p. 30
<slot>:.....see Screendump 9, p. 30
Conceptview creation:..... see Screendump 13, p. 35
<view>: ..... see Screendump 12, p. 35
Bridge creation:..... see Screendump 30, p. 62
<bridge>:..... see Screendump 29, p. 62
Instance creation:..... see Screendump 33, p. 68
<instance>:..... see Screendump 32, p. 67
Select objects: ..... see Screendump 26, p. 56



Screendump 7: Concept creation.

<concept>:.....see Screendump 5, p. 22
Create concept: .....see Screendump 6, p. 23
Cancel:.....see Screendump 3, p. 14

API

(**tr-concept-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a concept, `()` otherwise.

(**tr-create-concept** *conceptname key*) function in [libtrp]

-> concept, *conceptname* $\text{string}_{nas,adk}$,

key ((*name* string_{nas} *type* string|concept_{ivt} constructor $\text{string}_{nas,ivc}$ nature $\text{string}_{nas,ivn}$)$_{ivks+}$)$_{ivks}$, esk

Creates and returns a new concept of name *conceptname* and the provided set of concept fields specifications (*key*) as key. It will therefore be known of the system until deletion or initialisation.

(**tr-find-concept** *conceptname*) function in [libtrp]

-> concept, *conceptname* string_{nas}

Retrieves and returns the concept of name *conceptname*. When no such concept is known, returns `()`.

(**tr-concept** *conceptname key*) macro in [libtrp]

-> concept, *conceptname* string_{nas},

key ((*name* string_{nas} *type* string|concept_{ivt} constructor $\text{string}_{nas,ivc}$ nature $\text{string}_{nas,ivn}$)$_{ivks+}$)$_{ivks,\ esk}$

Retrieves and returns the concept of name *conceptname*. When no such concept is known, creates it (with the key specification *key*) and returns it.

(**tr-delete-concept** *concept*) function in [libtrp]

-> boolean, *concept* $\text{concept}_{nak,\ neo}$

Deletes the concept *concept* from the environment. This can only happen if the concept is as empty as at its creation. Afterwards, the concept does not exist anymore in the environment.

(**tr-erase-concept** *concept*) function in [libtrp]

-> boolean, *concept* $\text{concept}_{nak,\ neo}$

Deletes the concept *concept* and its content from the environment. Afterwards, neither the concept nor the conceptviews, conceptslots, bridges and objects it contains exist anymore in the environment.

(**tr-concept-name** *concept*) function in [libtrp]

-> string, *concept* concept_{nak}

Returns the name of the concept *concept*.

(**tr-concept-rename** *concept name*) function in [libtrp]

-> string, *concept* concept_{nak}, *name* $\text{string}_{nas,\ adk}$

Changes the name of the concept *concept* for the string *name*. This also changes the names of the root classes of each concept viewpoint if any.

(**tr-concept-key** *concept*) function in [libtrp]

-> (keyslot*), *concept* concept_{nak}

Returns the key of the concept *concept*, i.e. the (ordered) list of keyslots which forms the key.

Known bug: this is currently a set of string (the names of the key slots) which is returned.

(**tr-concept-slots** *concept*) function in [libtrp]

-> { conceptslot* }, *concept* concept_{nak}

Returns the set of concept slots of the concept *concept*.

(**tr-concept-sorted-slots** *concept optional: sort-function*) function in [libtrp]

-> { conceptslot* }, *concept* concept_{nak}, *sort-function* function

Returns the set of concept slots of the concept *concept* in their creation order or in their order in the base description file by default. If the *sort-function* is provided, the concept slots are sorted according to it.

(**tr-concept-views** *concept*) function in [libtrp]

-> { conceptview* }, *concept* concept_{nak}

Returns the set of concept views of the concept *concept*.

(**tr-concept-sorted-views** *concept optional: sort-function*) function in [libtrp]

-> { conceptview* }, *concept* concept_{nak}, *sort-function* function

Returns the set of concept views of the concept *concept* in their creation order or in their order in the base description file by default. If the *sort-function* is provided, the concept views are sorted according to it.

(**tr-concept-bridges** *concept*) function in [libtrp]

-> { bridge* }, *concept* $\text{concept}_{\text{nak}}$

Returns the set of bridges across viewpoints of the concept *concept*.

(**tr-concept-instances** *concept*) function in [libtrp]

-> { instance* }, *concept* $\text{concept}_{\text{nak}}$

Returns the set of instances of the concept *concept*.

(**tr-concept-sorted-instances** *concept optional: sort-function*) function in [libtrp]

-> { instance* }, *concept* $\text{concept}_{\text{nak}}$, *sort-function* function

Returns the set of instances of the concept *concept* in their creation order or in their order in the base description file by default. If the *sort-function* is provided, the objects are sorted according to it.

Examples

```
? (tr-create-concept "agent"
   '(("name" "chaine" "un" "propriete"))
     ("subsidiary" "chaine" "un" "propriete")))
= (tr-concept "agent")
? (tr-find-concept "agent")
= (tr-concept "agent")
? (tr-concept-p (tr-find-concept "agent"))
= (tr-concept "agent")
? (tr-delete-concept (tr-concept "agent"))
= t
? (tr-concept-p ())
= ()
? (tr-create-concept "agent" '(("name" "chaine" "un" "propriete")))
= (tr-concept "agent")
? (tr-add-concept (tr-current-base) (tr-find-concept "agent"))
= ((tr-concept "agent"))
? (tr-base-concepts (tr-current-base))
= ((tr-concept "agent"))
? (tr-concept-name (tr-find-concept "agent"))
= "agent"
? (tr-concept-key (tr-find-concept "agent"))
= ("name")
? (tr-concept-slots (tr-find-concept "agent"))
= ((tr-conceptslot (tr-concept "agent") "name"))
? (tr-concept-views (tr-find-concept "agent"))
= ()
? (tr-concept-views (tr-find-concept "agent"))
? (tr-concept-bridges (tr-find-concept "agent"))
= ()
? (tr-concept-instances (tr-find-concept "agent"))
= ()
? (tr-concept-instances (tr-find-concept "agent"))
= ()
```

Field definition: conceptual aspects

Field definitions assert that the instances of a concept have a particular set of fields. They also assert important characteristics of the fields such as the type of the field value, the fact that it is a property or a composition link. They thus define the structure of the objects. The definition of a concept field is unique for the whole concept. It contains the minimum information for TROEPS to handle the field values of individual objects. It is valid under each viewpoint in which it can only be refined. The definition of a concept field appears in the definition of the concepts; for a field to appear in a class, it must have been defined in the concept first; conversely, a field can appear in a concept and remain hidden under every viewpoint. It is thus useless for manipulation by the user of the knowledge base but participates in the essence of the concept. The characteristics of the concept field are:

- The base type of the values of the fields;
- The constructor of the type if necessary;
- The nature of the field;
- Computation mechanisms.

Each of these characteristics are introduced by what is called a descriptor. As an example the `size` field of a `house` is a property (`propriete` nature), has only one (`un` constructor) value which is an `integer` (type) and can be computed by summing the size of the `rooms` field which is a composite (`component` nature) field whose value is a set (`ensemble` constructor) of objects in the `room` concept (type). All is detailed below.

Type

The type descriptor for a concept introduces the basic value type of a particular field. There are two important kinds of field values in TROEPS: objects (instances of a TROEPS concept) and values (of a particular basic type). In addition, the field value itself can be a homogeneous collection of objects or values. This is indicated by the constructor descriptor (see below). When the basic value must be an object, the value of the type descriptor is the concept to which that object must belong. When it is a value, the value of the type descriptor must be the type of the value. Types must be distinguished between elementary types and record types.

Elementary types are defined in the type system through abstract data types. TROEPS is provided with a basic set of types (`entier`, `chaine`, `booleen`, `reel`) through the `trtdum` module. The user can extend this set very easily (see Appendix 5).

Record types are automatically created by the type system in order to help many tasks of the TROEPS system. They are accessible as types by the user. Thus, it is possible to declare a field as having a value of a record type. However, this record type must correspond to the type associated with a concept. The difference between an object, instance of a concept, and a value of the record-type corresponding to that concept lies in the identity-integrity mechanism. As said above, TROEPS ensures that two objects with the same key cannot be created; this is no longer true for the record values. Moreover, a concept records the set of its instances while a type does not record the set of its values. When queried for the object with a particular key, a concept will return the only object of that kind if it has been created while a type queried for a value with particular field values will return a new record with these characteristics regardless to whether an equal value has been created before or not. Thus, the same query for the same object returns the same object (identity) while the same query for the same value returns two

different values (equality but not identity). As a consequence, it is up to the user (or his program) to record the values which cannot be found by the type system.

This is generalised to the object/value difference without regarding the fact that the value is a record, a list or any other kind of value: the object is identified with its concept and its key while the values are identified by their structure. This leads to important consequences on keys: an object can have either another object or a value as the value of a field of its key, but they are treated differently by TROEPS:

- If the object is modified this does not matter since, it is always identifiable by its key (which cannot be modified).
- The value cannot be modified (either adding a value to a set, changing a field of a record or incrementing an integer) because this would be interpreted as changing it to another value and thus changing the key.

Constructor

The constructor descriptor indicates whether the value of the field will be a unique entity (of the type given by the type descriptor), a list of entities (of the type given by the type descriptor) or a set of entities (of the type given by the type descriptor). The value of the descriptor is a keyword which can be `un`, `liste` (list) or `ensemble` (set).

A list is a (possibly empty) ordered collection of entities which can appear more than once. A set is a (possibly empty) collection of entities with no repetition. A set is not ordered (thus there is no first nor n^{th} element of a set).

Since these collection constructors are pre-defined in TROEPS, TROEPS provides field accessors for adding elements, removing, testing membership of entities to these fields. Its is also possible to reach the value as a whole and to apply classical TALK operators on it (see the "object" section).

Nature

Concept fields are distinguished by their nature. The pre-defined natures for the time being are `propriete` (property), `composant` (component) and `lien` (link). These are the values admissible for the nature descriptor. The nature covers quite deep semantic functions of the system and thus some important reflection have to be carried out before achieving a satisfying extendible nature system.

The property is not attached a particular meaning except that the field defines an object or value which is that of the corresponding property of the object. By opposition, the component nature is attached to fields with a very precise semantics.

The link nature is a mean to define recursive links between objects. The links introduce restrictions with regard first class properties: they are not taken into account by classification and filtering algorithms. All the other functions (e.g. type checking when assigning a link or migrating the object) are available.

The study of the composition relation has shown that it contains a plethora of meanings and could eventually be described as a taxonomy emphasising functional, homeomerous and separable characteristics. It has shown six different relations: component/integral object (handle-cup), member/collection (tree-forest), portion/mass (slice-pie), stuff/object (steel-bike), feature/activity (paying-shopping) and place/area (oasis-desert). The composition relation is *functional* when the parts support a functional role with respect to the whole; it is *homeomerous* when the parts are similar to each other and to the whole to which they belong and it is

separable when the parts can be separated from the whole to which they are connected. In TROEPS, the component/object meaning is used. However, this meaning is so rich that no checking about it can be made by the system. It is up to the designers of knowledge bases to take care of this, so that the reasoning they might ask the system to perform is valid.

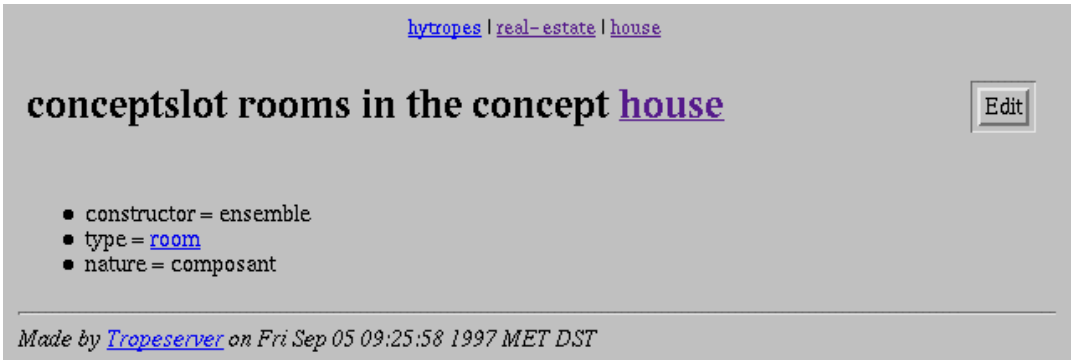
The services provided by TROEPS for component fields are the following:

- Existence: whenever an object with a component field is created (resp. destroyed), the equivalent value (component) is assumed to exist. This does not prevent, however, an object of the component type from being created independently.
- Non-cyclicity: an object cannot be (directly or indirectly) the component of one of its components.
- Non sharability: the component of an object cannot be component of another object except if both objects are also related by (the transitive closure of) composition fields. This does not prevent it to be referred to as the value of property or link fields of other objects.

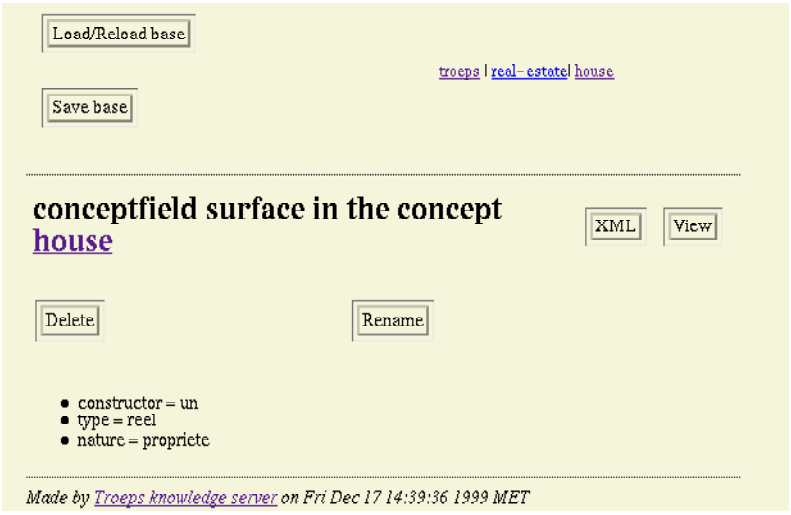
Since composition is modelled thanks to composite fields, a same object can be decomposed in many ways, as many as there are viewpoints. For instance, a car can be composed of the engine, the breaking system, the transmission system, etc., in a mechanical viewpoint, and can also be composed of its doors, its body, etc., in a physical viewpoint. On the contrary, one can wish to decompose an object in the same way in every viewpoint; this can be done by defining composite fields common to all the viewpoints.

The transitive reduction of component relations under a viewpoint constitutes a tree of exclusive branches. The exclusivity of the component nature means that an object cannot be a component of two different objects unless they are themselves in a component relation.

Interface

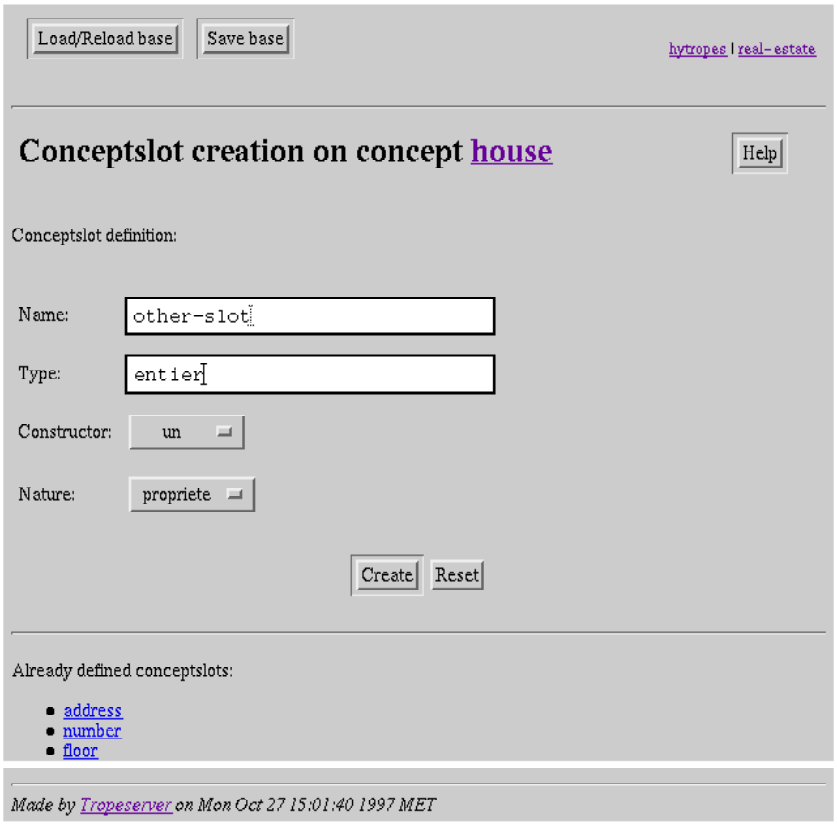


Screendump 8: Concept field display.
Edit: .....see Screendump 9, p. 30
type: .....see Screendump 5, p. 22



Screendump 9: Concept field edition.

View:..... see Screendump 11, p. 34
type:..... see Screendump 5, p. 22



Screendump 10: Concept field creation.

Create: ..... see Screendump 6, p. 23
Reset: ..... see Screendump 10

API

(**tr-conceptslot-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a concept field, `()` otherwise.

(**tr-create-conceptslot** *concept conceptslotname type const nature*) function in [libtrp]

-> conceptslot, *concep*t concept$_{nak}$,
conceptslotname string$_{nas,adks}$, *type* string|concept$_{ivt}$, *const* string$_{nas,\ ivc}$, *nature* string$_{nas,\ ivn}$

Creates and returns a new concept field of name *conceptslotname* in the concept *concept*. It will therefore be known of the system until deletion or initialisation. The conceptslot is automatically added to the list of fields of the concept *concept* whose type is updated.

(**tr-find-conceptslot** *concept conceptslotname*) function in [libtrp]

-> conceptslot, *concep*t concept$_{nak}$, *conceptslotname* string$_{nas}$

Retrieves and returns the concept field of name *conceptslotname* in the concept *concept*. When no such concept field is known, returns `()`.

(**tr-conceptslot** *concept conceptslotname type const nature*) macro in [libtrp]

-> conceptslot, *concep*t concept$_{nak}$, *conceptslotname* string$_{nas}$,
type string|concept$_{ivt}$, *const* string$_{nas,\ ivc}$, *nature* string$_{nas,\ ivn}$

Retrieves and returns the concept field of name *conceptslotname* in the concept *concept*. When no such concept field is known, creates and returns it.

(**tr-delete-conceptslot** *conceptslot*) function in [libtrp]

-> boolean, *conceptslot* conceptslot$_{naks,cmk,\ neo}$

Deletes the concept field *conceptslot*. This can only happen if the concept slot is as empty as at its creation and no classslot with the same name exists in the concept.

(**tr-erase-conceptslot** *conceptslot*) function in [libtrp]

-> boolean, *conceptslot* conceptslot$_{naks,cmk,\ neo}$

Deletes the concept field *conceptslot* from the environment. Afterwards neither the conceptslot nor the classslots refering to it nor the fields of the same name in the objects exist anymore in the environment.

(**tr-conceptslot-concept** *conceptslot*) function in [libtrp]

-> concept, *conceptslot* conceptslot$_{naks}$

Returns the concept of the concept field *conceptslot*.

(**tr-conceptslot-name** *conceptslot*) function in [libtrp]

-> string, *conceptslot* conceptslot$_{naks}$

Returns the name of the concept field *conceptslot*.

(**tr-conceptslot-rename** *conceptslot name*) function in [libtrp]

-> string, *conceptslot* conceptslot$_{naks}$, *name* string$_{nas,\ adks}$

Changes the name of the concept field *conceptslot* for the string *name*. This also changes the name of all the corresponding class fields.

(**tr-conceptslot-viewpoints** *conceptslot*) function in [libtrp]

-> { conceptview* }, *conceptslot* conceptslot$_{naks}$

Returns the set of concept viewpoints under which the field is visible in its concept. This set is automatically updated when a class field is declared in a class.

(**tr-get-conceptslot-type** *conceptslot*) function in [libtrp]

-> string, *conceptslot* conceptslot$_{naks}$

Returns the type of the concept field *conceptslot*.

(**tr-get-conceptslot-constructor** *conceptslot*) function in [libtrp]

-> string, *conceptslot* conceptslot$_{naks}$

Returns the constructor of the concept field *conceptslot*.

(**tr-get-conceptslot-nature** *conceptslot*) function in [libtrp]

-> string, *conceptslot* conceptslot$_{naks}$

Returns the nature of the concept field *conceptslot*.

Examples

```
? (tr-create-conceptslot (tr-find-concept "agent") "segment" "chaine" "un" "propriete")
= (tr-conceptslot (tr-concept "agent") "segment")
? (tr-create-conceptslot (tr-find-concept "agent")
    "houses" (tr-find-concept "house") "ensemble" "propriete")
= (tr-conceptslot (tr-concept "agent") "houses")
? (tr-create-conceptslot (tr-find-concept "agent") "subsidiary" "chaine" "un" "propriete")
= (tr-conceptslot (tr-concept "agent") "subsidiary")
? (tr-concept-sorted-slots (tr-find-concept "agent"))
= ((tr-conceptslot (tr-concept "agent") "name")
   (tr-conceptslot (tr-concept "agent") "segment")
   (tr-conceptslot (tr-concept "agent") "houses")
   (tr-conceptslot (tr-concept "agent") "subsidiary"))
? (tr-delete-conceptslot (tr-conceptslot (tr-concept "agent") "subsidiary"))
= t
? (tr-concept-slots (tr-find-concept "agent"))
= ((tr-conceptslot (tr-concept "agent") "houses")
   (tr-conceptslot (tr-concept "agent") "name")
   (tr-conceptslot (tr-concept "agent") "segment"))
? (tr-create-conceptslot (tr-find-concept "agent") "area" "chaine" "un" "propriete")
= (tr-conceptslot (tr-concept "agent") "area")
? (tr-create-conceptslot (tr-find-concept "agent") "comission" "reel" "un" "propriete")
= (tr-conceptslot (tr-concept "agent") "comission")
? (tr-find-conceptslot (tr-find-concept "agent") "houses")
= (tr-conceptslot (tr-concept "agent") "houses")
? (tr-conceptslot-p (tr-find-conceptslot (tr-find-concept "agent") "name"))
= (tr-conceptslot (tr-concept "agent") "name")
? (tr-conceptslot-p ())
= ()
? (tr-conceptslot-concept (tr-conceptslot (tr-find-concept "agent") "segment"))
= (tr-concept "agent")
? (tr-conceptslot-name (tr-conceptslot (tr-find-concept "agent") "segment"))
= "segment"
? (tr-get-conceptslot-type (tr-conceptslot (tr-find-concept "agent") "segment"))
= "chaine"
? (tr-get-conceptslot-type (tr-conceptslot (tr-find-concept "agent") "houses"))
= (tr-concept "house")
? (tr-get-conceptslot-constructor (tr-conceptslot (tr-find-concept "agent") "segment"))
= "un"
? (tr-get-conceptslot-constructor (tr-conceptslot (tr-find-concept "agent") "houses"))
= "ensemble"
? (tr-get-conceptslot-nature (tr-conceptslot (tr-find-concept "agent") "segment"))
= "propriete"
```

Viewpoints

Individual instances of a concept can be seen under different viewpoints — and thus described — from different viewpoints; each viewpoint determines a specific class hierarchy, and hides some characteristics of a concept (field and class constraints). In such a way, it is possible to see only the fields concerned with a viewpoint and to structure the concept in a class hierarchy which is significant for the viewpoint. The observation of an object through a viewpoint reveals only a part of its fields. An instance is observable under each viewpoint, but only a sub-set of its fields. However, the fields of the key are always visible in each viewpoints in order to be able to identify the objects.

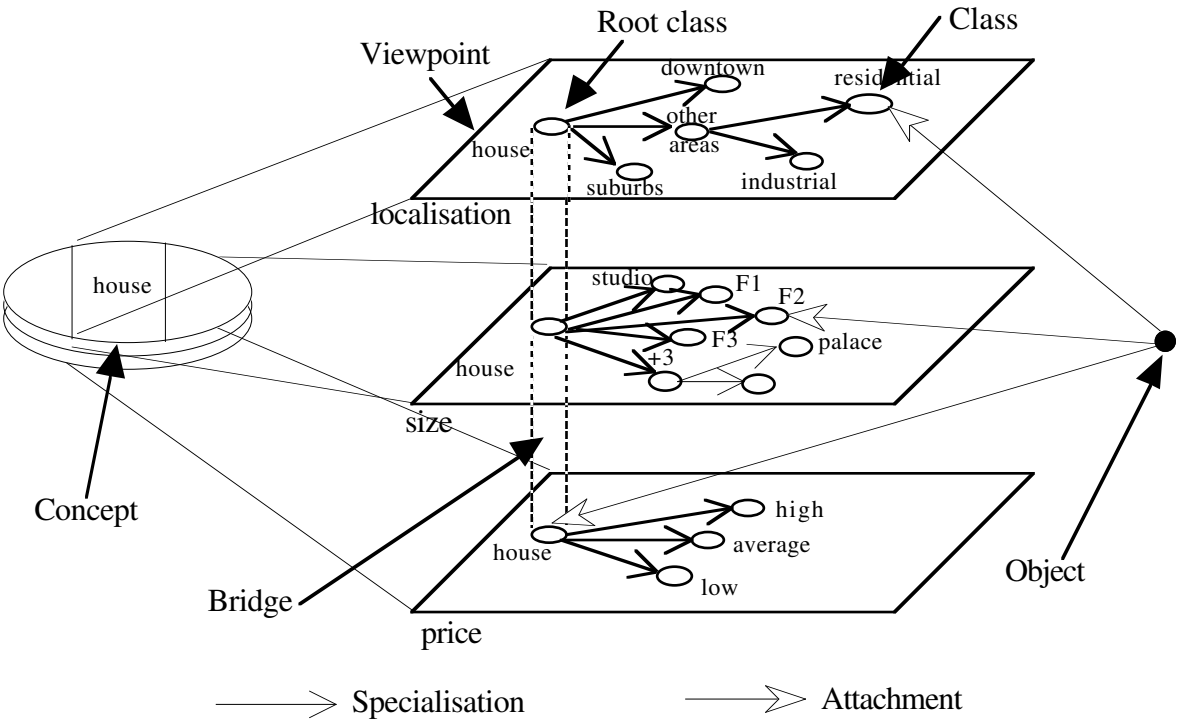


Figure 6: The notions involved in a TROEPS knowledge base.

Structure of a viewpoint

A viewpoint defines a particular decomposition of the set of objects instance of the concept. This is materialised by a particular taxonomy of classes proper to the viewpoint. In a viewpoint, classes are related by the specialisation relation. A class (so called super-class) is specialised in more restrictive classes (its sub-classes) which represent sub-sets of the objects attached to the super-class. The specialisation relation is transitive: a class is a sub-class of its direct super-class and of its super-classes. Every class is described inside the viewpoint (see the "class" section).

This class taxonomy in a viewpoint has a tree structure. The root represents all the objects of the concept, it has no super-class. The root class is named after the concept. The root class cannot be constrained (because this could exclude some instances of the concept). Any other class in the hierarchy has only one direct super-class.

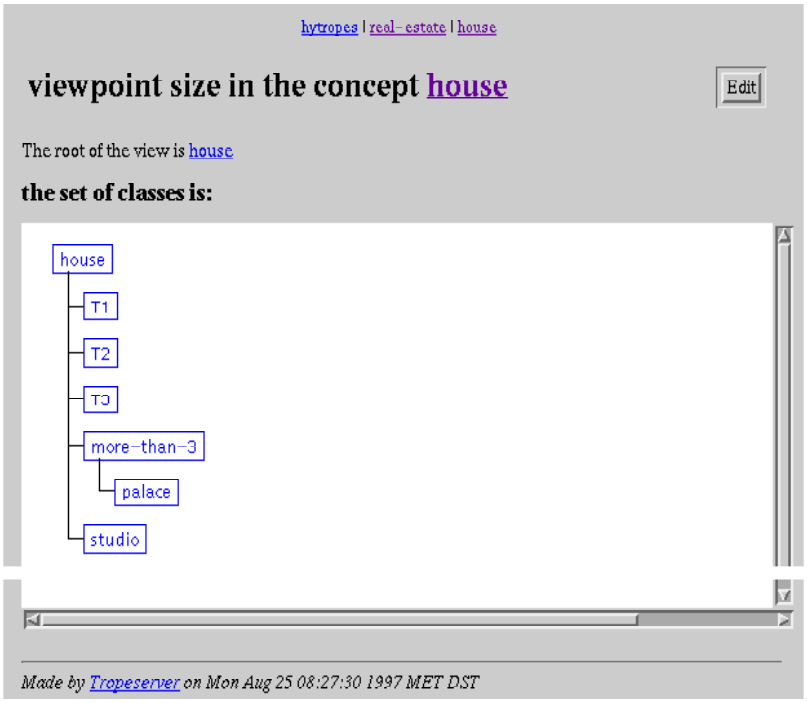
In each viewpoint, objects are attached to one and only one class of the class hierarchy, in which each class represents a set of objects. These viewpoints on a concept are not totally independent: they communicates through bridges (see the “bridge” section).

Constraints on the taxonomy

By default, in TROEPS, taxonomies do not have particular constraints. The classes are descriptions of (i.e. necessary conditions for belonging to) an intended set of objects but they are not constrained to be exclusive nor exhaustive. An exclusive taxonomy is a taxonomy in which the direct sub-classes of a particular class have pairwise disjoint extensions: an object cannot be member of two of these classes. An exhaustive taxonomy is one in which the direct sub-classes of a particular class cover (as a whole) the extension of the super-class: any object belonging to the super-class is a member of one of its sub-classes. Lastly, a definitional taxonomy is one for which the extension of the class exactly corresponds to the set of objects which comply the definition of the class: the description expresses necessary and sufficient conditions for belonging to the class.

These defaults can be changed viewpoint by viewpoint. Then TROEPS algorithms take advantage of the indications in order to provide more appropriate results.

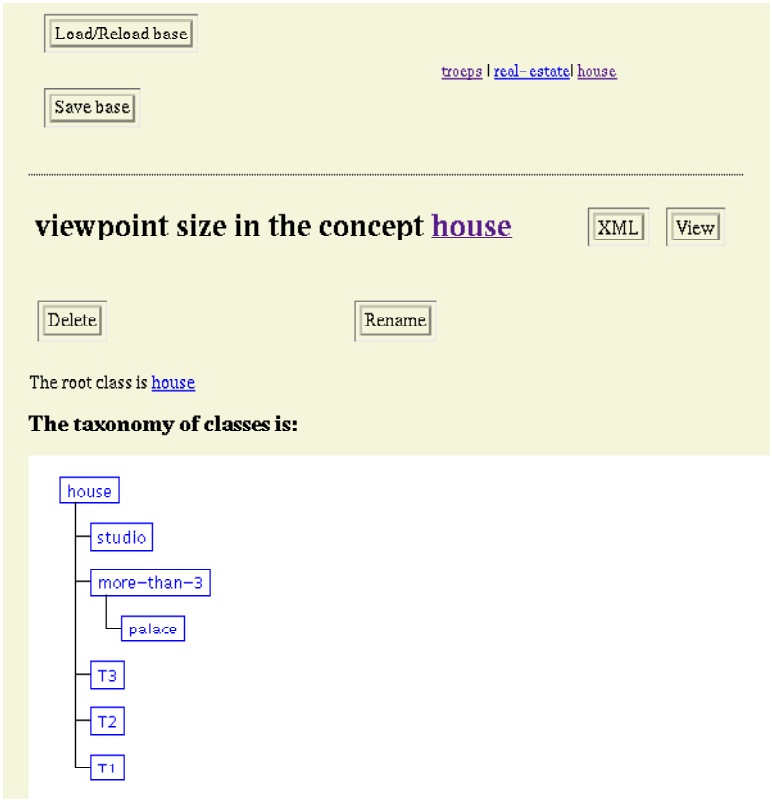
Interface



Screendump 11: Viewpoint display.

Edit: ..... see Screendump 12, p. 35

<class> (dble-click):..... see Screendump 14, p. 40



Screendump 12: Viewpoint edition.

View:..... see Screendump 11, p. 34
<class> (dble-click):..... see Screendump 15, p. 41

Screendump 13: Viewpoint creation.

Create: ..... see Screendump 12, p. 35
Reset: ..... see Screendump 13
<conceptview>: ..... see Screendump 11, p. 34

API

(tr-conceptview-p *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a concept viewpoint, `()` otherwise.

(**tr-create-conceptview** *concept conceptviewname*) function in [libtrp]

-> conceptview, *concept* concept$_{nak}$, *conceptviewname* string$_{nas,adkv}$

Creates and returns a new concept viewpoint of name *conceptviewname* in the concept *concept*. It will therefore be known of the system until deletion or initialisation.

(**tr-find-conceptview** *concept conceptviewname*) function in [libtrp]

-> conceptview, *concept* concept$_{nak}$, *conceptviewname* string$_{nas}$

Retrieves and returns the concept viewpoint of name *conceptviewname* in the concept *concept*. When no such concept viewpoint is known, returns `()`.

(**tr-conceptview** *concept conceptviewname*) macro in [libtrp]

-> conceptview, *concept* concept$_{nak}$, *conceptviewname* string$_{nas}$

Retrieves and returns the concept viewpoint of name *conceptviewname* in the concept *concept*. When no such concept viewpoint is known, creates and returns it.

(**tr-delete-conceptview** *conceptview*) function in [libtrp]

-> boolean, *conceptview* conceptview$_{nakv, neo}$

Deletes the concept viewpoint *conceptview* from its concept. This can only happen if the concept view is as empty as at its creation. Afterwards, the concept viewpoint does not exist anymore in the environment.

(**tr-erase-conceptview** *conceptview*) function in [libtrp]

-> boolean, *conceptview* conceptview$_{nakv, neo}$

Deletes the concept viewpoint *conceptview* and its content from its concept. Afterwards, neither the concept viewpoint nor the classes it contains nor the bridges which refer to these classes exist anymore in the environment.

(**tr-conceptview-concept** *conceptview*) function in [libtrp]

-> concept, *conceptview* conceptview$_{nakv}$

Returns the concept of the concept viewpoint *conceptview*.

(**tr-conceptview-name** *conceptview*) function in [libtrp]

-> string, *conceptview* conceptview$_{nakv}$

Returns the name of the concept viewpoint *conceptview*.

(**tr-conceptview-rename** *conceptview name*) function in [libtrp]

-> string, *conceptview* conceptview$_{nakv}$, *name* string$_{nas, adkv}$

Changes the name of the concept view *conceptview* for the string *name*.

(**tr-conceptview-root** *conceptview*) function in [libtrp]

-> class, *conceptview* conceptview$_{nakv}$

Returns the root class of the concept viewpoint *conceptview*.

(**tr-conceptview-exhaustive** *conceptview optional: boolean*) function in [libtrp]

-> boolean, *conceptview* conceptview$_{nakv}$, *boolean* boolean

Sets (if *boolean* is provided) the exhaustiveness of the concept viewpoint *conceptview* to *boolean* and returns the exhaustiveness of the concept viewpoint.

(**tr-conceptview-exclusive** *conceptview optional: boolean*) function in [libtrp]

-> boolean, *conceptview* conceptview$_{nakv}$, *boolean* boolean

Sets (if *boolean* is provided) the exclusiveness of the concept viewpoint *conceptview* to *boolean* and returns the exclusiveness of the concept viewpoint.

(**tr-conceptview-definitional** *conceptview optional: boolean*) function in [libtrp]

-> boolean, *conceptview* conceptview$_{nakv}$, *boolean* boolean

Sets (if *boolean* is provided) the definitional character of the concept viewpoint *conceptview* to *boolean* and returns the value of the definitional character of the concept viewpoint.

Examples

```
? (tr-create-conceptview (tr-concept "agent") "buyer")
= (tr-conceptview (tr-concept "agent") "buyer")
? (tr-create-conceptview (tr-concept "agent") "seller")
= (tr-conceptview (tr-concept "agent") "seller")
? (tr-conceptview-p (tr-find-conceptview (tr-concept "agent") "buyer"))
= (tr-conceptview (tr-concept "agent") "buyer")
? (tr-conceptview-p ())
= ()
? (tr-create-conceptview (tr-concept "agent") "tax-collector")
= (tr-conceptview (tr-concept "agent") "tax-collector")
? (tr-delete-conceptview (tr-conceptview (tr-concept "agent") "tax-collector"))
= t
? (tr-concept-views (tr-find-concept "agent"))
= ((tr-conceptview (tr-concept "agent") "buyer")
   (tr-conceptview (tr-concept "agent") "seller"))
? (tr-concept-sorted-views (tr-find-concept "agent"))
= ((tr-conceptview (tr-concept "agent") "buyer")
   (tr-conceptview (tr-concept "agent") "seller"))
? (tr-conceptview-concept (tr-conceptview (tr-concept "agent") "buyer"))
= (tr-concept "agent")
? (tr-conceptview-name (tr-conceptview (tr-concept "agent") "buyer"))
= "buyer"
? (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
? (tr-conceptview-exhaustive (tr-conceptview (tr-concept "agent") "buyer"))
= ()
? (tr-conceptview-exclusive (tr-conceptview (tr-concept "agent") "buyer"))
= ()
? (tr-conceptview-exclusive (tr-conceptview (tr-concept "agent") "buyer") t)
= t
? (tr-conceptview-exclusive (tr-conceptview (tr-concept "agent") "buyer"))
= t
? (tr-conceptview-definitional (tr-conceptview (tr-concept "agent") "buyer"))
= ()
```

Class definition

A class specifies a virtual set of instances in a concept. This set is described with the help of restrictions on fields of the objects.

The class structure

A class is identified by the viewpoint in which it is defined (which in turns refers to its concept), a name (unique in a concept viewpoint) and a super-class. These three elements are mandatory for all classes but the root classes. Root classes are automatically defined when a viewpoint is defined on a concept. They have no super-class and their name is that of the concept.

A class can introduce field restrictions on fields visible under the viewpoint, this constitutes the set of class fields of a class. The field restrictions are considered in the next section. A root class, however, cannot introduce any restriction on the fields because it is meant to represent the whole set of instances of the concept.

Last, a class can introduce bridges from it (and other classes in other viewpoints) to another class (in another viewpoint). Bridges are examined below. A root class is automatically connected through bridges with the root class of any other viewpoint of the concept.

Specialisation

The specialisation relation between classes support several interpretations whose discussion is out of the scope of this manual. It is expressed by the transitive closure of the super-class—sub-class link: the sub-class is a specialisation of the super-class and also specialisation of the classes its super-class is a specialisation of (see example). A class can have only one super-class and the root class is the only class which does not have a super-class. A class can have several sub-classes.

The specialisation is intended to be exclusive: in the same viewpoint, a particular instance cannot be a member of two classes which are not in a specialisation relation. However, the exclusivity is only subjective, this means that the system cannot warrant it from the definition of the classes. Of course, the system will signal if one tries to attach an object to two incomparable classes but it will not be able to decide whether the two classes are really exclusive. For example, some people once thought that in the person classes, republican and Quakers were two exclusive classes since republican were reputed to be bellicists and Quakers pacifists. However, it turned out that Richard Nixon was both republican and Quaker: he thus must be attached (who said “should have been”?) to two supposedly exclusive classes.

On the other hand, TROEPS does not suppose that sub-classes are exhaustive (i.e. that any member of a class must be a member of a sub-class). These two properties are very important for the determination of the classification algorithms which can take advantage of the properties. They are also used by other mechanisms (such as the definition of bridges, see below the “bridge” section).

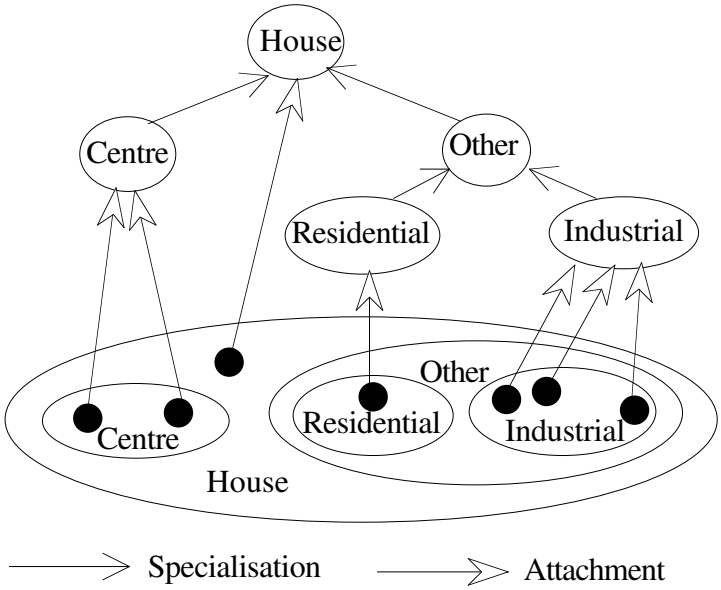


Figure 7: As an example, here, the class-sub-class link is established between Residential and Other, Industrial and Other, Other and House (the root class). This means that Residential (and Industrial as well) is a specialisation of both Other and House. This specialisation is interpreted as the inclusion of the set of members.

The specialisation constrains any object attached to a class to be also member of its super-classes and that any field definition in a class is more (or equally) restricted than that of its super-classes. Consequently, objects are only attached to their most specific class and the system is able to infer that they are member to all of its super classes. Field restrictions do also not have to be expressed completely. It is sufficient to define only the restrictions from the super-classes the system will be able to infer the complete type of the field from the super-classes type restrictions and the new one.

As an example, in the real estate base, the concept key is `address`, `floor` and `number`. Thus, under the `localisation` viewpoint, these three fields are visible. The `center` class is a subclass of the root class (called `house`) of this viewpoint. There are two alternatives for characterising the set of objects which can be classified under that class:

- Unhiding a new field called `area` with the possible values in the `center` restricted as {`"grenoble"`}.
- Restricting the addresses to be addresses in the center area (see below for field restrictions in classes).

Any `house` which is member a of the `center` class will also be a member of the `house` class (of which all houses are members).

Membership and attachment

An object is attached to only one class in each viewpoint. It is also said that the object is member of all super-classes of that class. As a matter of fact, it is compliant with the field restriction on that class (and thus with the field restriction of all the super-classes of that class).

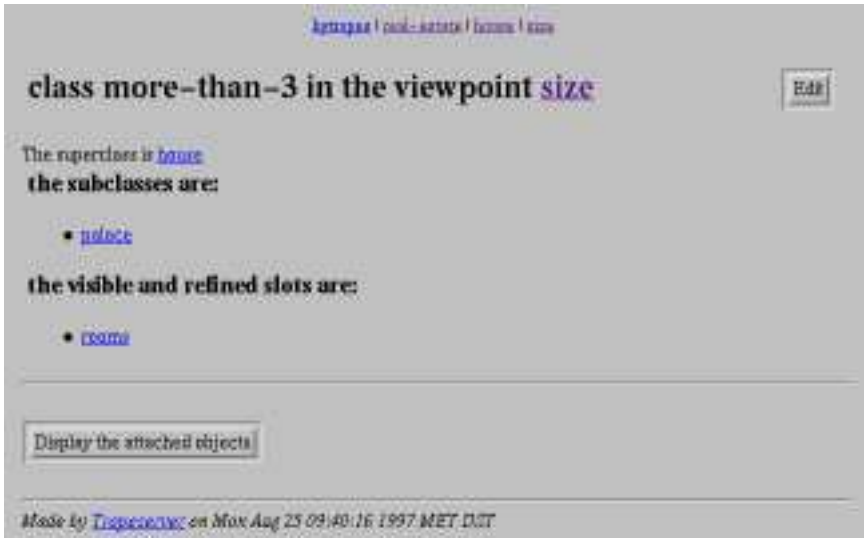
So, while objects are *instances* of a concept, they can only be *attached* to a class. Upon creation, each instance is attached to each root class of each viewpoint of the concept. The difference is critical for several reasons:

- The objects can change their attachment class from time to time while they are always instances of the same concept;

- The objects can violate the constraints on their classes (they will be re-classified under the root class) but cannot violate those of the concepts;
- The concept refers to methods allowing to compute the true value of a field while a class only refers to methods for guessing a hypothetical value for a field.

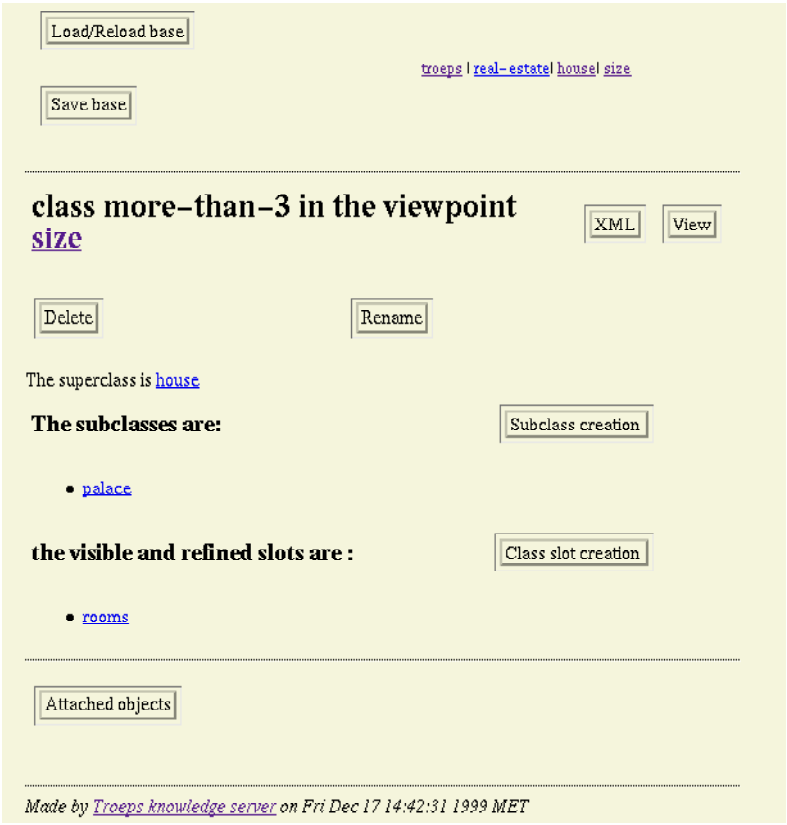
The attachment of an object to a class is a voluntary act from the users (or one of their programs). This act can be accomplished even if some fields are missing. The only constraint is that available field values satisfy their types in the attachment class. However, the consequence of attachment is that unfilled fields must implicitly satisfy their type as well even if they are unknown.

Interface



Screendump 14: Class display.

Edit: ..... see Screendump 15, p. 41
<class>: ..... see Screendump 14
<slot>:..... see Screendump 19, p. 48
Display the attached objects:..... see Screendump 16, p. 41

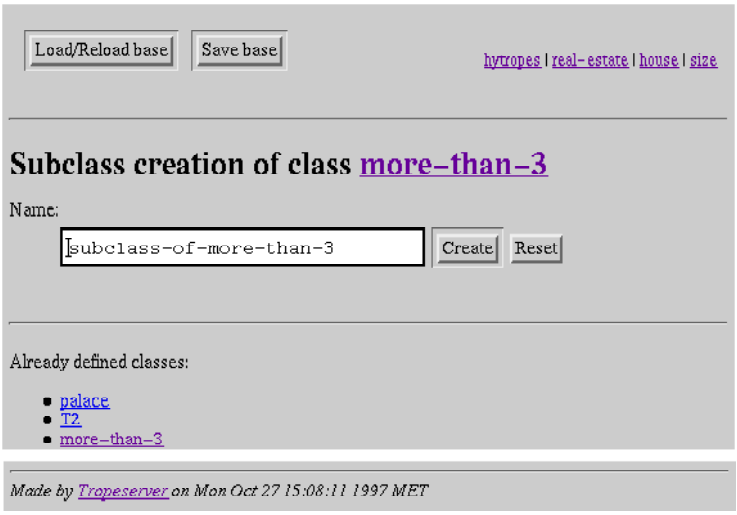


Screendump 15: Class edition.

View:..... see Screendump 14, p. 40
Subclass creation:..... see Screendump 16, p. 41
<class>: ..... see Screendump 15
Class slot creation:..... see Screendump 25, p. 51
<slot>:..... see Screendump 19, p. 48
Display the attached objects:..... see Screendump 16, p. 41

Screendump 16: Attached objects.

<object>:..... see Screendump 31, p. 66



Screendump 17: Class creation.

Create: ..... see Screendump 15, p. 41
Reset: ..... see Screendump 17
<class>: ..... see Screendump 14, p. 40

API

(**tr-class-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a class, `()` otherwise.

(**tr-create-class***superclass classname*) function in [libtrp]

-> class,, *superclass* $\text{class}_{\text{nac}}$ *classname* $\text{string}_{\text{nas, adc}}$

Creates and returns a new class of name *classname*. This class is a specialisation of the class *superclass* and is part of the concept viewpoint of this last class. It will, therefore, be in the concept viewpoint hierarchy until deletion or initialisation.

(**tr-find-class** *conceptview classnam*e) function in [libtrp]

-> class, *conceptview* $\text{conceptview}_{\text{nakv}}$, *classname* $\text{string}_{\text{nas}}$

Retrieves and returns the class of name *classname* in the concept viewpoint *conceptview*. When no such class is known, returns `()`.

(**tr-class** *conceptview classname superclass*) macro in [libtrp]

-> class, *conceptview* $\text{conceptview}_{\text{nakv}}$, *classname* $\text{string}_{\text{nas}}$, *superclass* $\text{class}_{\text{nac, nikv}}$

Retrieves and returns the class of name *classname* in the concept viewpoint *conceptview*. When no such class is known, creates it as a specialisation of the class *superclass* and returns it.

(**tr-delete-class** *class*) function in [libtrp]

-> boolean, *class* $\text{class}_{\text{nac, cdrc, neo}}$

Deletes the class *class* from the environment. This leads to its deletion in its concept viewpoint hierarchy and can only happen if the class is as empty as at its creation. Afterwards, the class does not exist anymore in the environment.

(**tr-erase-class** *class*) function in [libtrp]

-> boolean, *class* $\text{class}_{\text{nac, cdrc, neo}}$

Deletes the class *class* and its content from the environment. Afterwards, neither the class nor its classslots and constraints nor the bridges which refer to it exist anymore in the environment.

(**tr-class-conceptview** *class*) function in [libtrp]

-> conceptview, *class* $\text{class}_{\text{nac}}$

Returns the concept viewpoint of the class *class*.

(**tr-class-concept** *class*) function in [libtrp]

-> concept, *class* $\text{class}_{\text{nac}}$

Returns the concept of the class *class*.

(**tr-class-name** *class*) function in [libtrp]

-> string, *class* $\text{class}_{\text{nac}}$

Returns the name of the class *class*.

(**tr-class-rename** *class name*) function in [libtrp]

-> string, *class* $\text{class}_{\text{nac}}$, *name* $\text{string}_{\text{nas, adc, crrc}}$

Changes the name of the class *class* for the string *name*. The root class name cannot be changed.

(**tr-class-superclass** *class*) function in [libtrp]

-> class, *class* $\text{class}_{\text{nac}}$

Returns the superclass of the class *class* or `()` if *class* is a root class.

(**tr-class-subclasses** *class*) function in [libtrp]

-> { class* }, *class* $\text{class}_{\text{nac}}$

Returns the set of direct sub-classes of the class *class*.

(**tr-class-all-subclasses** *class*) function in [libtrp]

-> { class* }, *class* $\text{class}_{\text{nac}}$

Returns the set of sub-classes of the class *class*.

(**tr-class-superclass-p** *superclass class*) function in [libtrp]

-> boolean, *superclass* $\text{class}_{\text{nac}}$, *class* $\text{class}_{\text{nac, nikv}}$

Tells wether the class *superclass* is a superclass (at any level) of the class *class*.

(**tr-class-bridges-in** *class*) function in [libtrp]

-> { bridge* }, *class* $\text{class}_{\text{nac}}$

Returns the set of bridges whose destination is the class *class*.

(**tr-class-bridges-out** *class*) function in [libtrp]

-> { bridge* }, *class* $\text{class}_{\text{nac}}$

Returns the set of bridges whose source contains the class *class*.

(**tr-class-sorted-slots** *class optional: sort-function*) function in [libtrp]

-> { classslot* }, *class* $\text{class}_{\text{nac}}$, *sort-function* function

Returns the set of class fields of the class *class* in the creation order of the corresponding conceptslots (or their order in the base description file) by default. If the *sort-function* is provided, the classslots are sorted according to it.

(**tr-class-slots** *class*) function in [libtrp]

-> { classslot* }, *class* class$_{nac}$

Returns the set of class fields of the class *class*.

(tr-class-attached *class*) function in [libtrp]

-> { object* }, *class* class$_{nac}$

Returns the set of objects attached to the class *class*.

(tr-class-members *class*) function in [libtrp]

-> { object* }, *class* class$_{nac}$

Returns the set of objects member of the class *class*.

Examples

```
? (tr-create-class
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer")) "good")
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")
? (tr-create-class
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer")) "bad")
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
? (tr-create-class
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "seller")) "good")
= (tr-class (tr-conceptview (tr-concept "agent") "seller") "good")
? (tr-create-class
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "seller")) "bad")
= (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")
? (tr-create-class
    (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "good") "favorite")
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
? (tr-create-class
    (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "good") "valuable")
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
? (tr-class-p ())
= ()
? (tr-class-p (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
? (tr-create-class
    (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "good") "not-so-good")
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "not-so-good")
? (tr-delete-class (tr-class (tr-conceptview (tr-concept "agent") "buyer") "not-so-good"))
= t
? (tr-superclass-p
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "good")
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "valuable"))
= t
? (tr-superclass-p
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
= ()
? (tr-superclass-p
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
   (tr-find-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-conceptview
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= (tr-conceptview (tr-concept "agent") "buyer")
? (tr-class-concept
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= (tr-concept "agent")
? (tr-class-name (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= "favorite"
? (tr-class-superclass
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")
? (tr-class-superclass
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer")))
```

```
= ()
? (tr-class-subclasses
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-subclasses
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer")))
= ((tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
? (tr-class-all-subclasses
    (tr-conceptview-root (tr-conceptview (tr-concept "agent") "buyer")))
= ((tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
? (tr-class-bridges-in
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-bridges-out
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-slots
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-attached
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
? (tr-class-members
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ()
```

Class field definition: data typing aspects/unary constraints

Class fields are used in order to refine the field definition of the members of a particular class. Some of these fields are devoted to restrict the type of the values of a field (see the "concept field" section for basic values given in concepts), some other to indicate that the fields is now visible under the class. The type refinement is obtained through the addition of field descriptors which are optional in the definition of a field. They have to be filled only if this helps restricting the set of acceptable values of the field.

Type refinement

There is also a type descriptor in a class field definition. It is used in order to refine the type descriptor used in the concept. As in the concept case, it must be distinguished between types and classes.

For types, the type descriptor can only contain a sub-type of the initial type: for instance, it can be the ASCII-string as a sub-type of string. The reader should report to the "type" section (Appendix 5) in order to fully understand and use types and sub-types. Record types have implicit subtypes which are expressed exactly as subclasses types (this is translated by the system).

For classes, the field is refined by specifying a set of classes (of the same concept but different viewpoints) the value (an object here) must be member of. The meaning of that set of classes is that the object to fill the field must be a member of each class in the set. Of course, these classes are classes of the concept and the set of classes can only contain classes which are subclasses of classes in the super-class type specification (by default, if there is no restriction, it is intended that the root class of each missing viewpoint is in the set).

Advanced type refinement

Advanced type refinement applies to fields whose values are objects. It allows to add constraints on the possible values of the fields. This is achieved by defining a sub-class of a particular class which is called filter (see above the "class" section) and to add more field constraints on that class. A filter is defined exactly as a class (with a super-class, a viewpoint and a name), it usually contains more field restrictions and even more inference methods than the initial class but:

- It does not appear in the specialisation graph of its viewpoint;
- No object is properly attached to a filter.

It defines the set of admissible values as the objects which are member of its super-classes and which satisfy the restrictions of the filter. Such a filter is different from a filter used as an inference method in the use which is made of it.

An example of filter is the houses located in the center with more than 3 rooms.

Enumeration

The domain descriptor enumerates the set of possible values that can be in the field value. As an example, the `center` class will restrict the `area` field value (whose type is `chaine`, i.e. string) to be "`grenoble`". In case of collection types, it is possible to specify a set of sets or a set of lists as a domain.

Exclusion

The except descriptor enumerates a set of values which cannot be the value of the field. For instance, the set of `address` in the `residential` class will exclude values among `"factory St."` and `"big garbage dr."`. In case of collection types, it is possible to exclude a set of sets or a set of lists from the possible values.

Unary constraints

The condition descriptor contains a list of unary predicates which must be satisfied by the values of the fields. For the definitions of unary constraints or predicates, see next chapter. There is no pre-defined constraints in TROEPS. These predicates are checked against the whole value of the field. The value of the field must thus satisfy all the given predicates. For example, if the numbers in a street are (as in some European towns) odd on one side and even on the other, it is possible to delimit the district only by the numbers which are odd in a street (it is sometime usual to have one side in a district and the other side on the other, guess what? the "bec-de-gaz" are different from side to side).

Intervals

The intervals descriptor applies to ordered types. Types such as integer, real and strings are ordered. It is also possible to define order on other types and to define ordered concepts. The value of such a descriptor is a list of TROEPS interval structures. This restriction means that the field value must belong to one of the given intervals. As an example, under the `price` viewpoint, the `average` class constrains the `rent` field to be in the interval `{[10000. 15000.]}`.

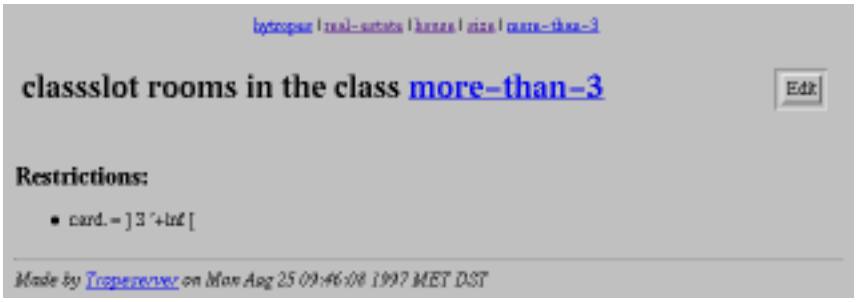
As an example, these intervals are obtained by the following TALK instructions:

```
[45 52]  => (tr-make-interval 45 t 52 t)
[45 52[  => (tr-make-interval 45 t 52 ())
]45 52]  => (tr-make-interval 45 () 52 t)
]45 52[  => (tr-make-interval 45 () 52 ())
]-∞ +∞[  => (tr-make-interval '-inf () '+inf ())
```

Cardinality

The cardinal descriptor only applies to collection types. It allows to restrict the cardinal (for a set and length for a list) to be a particular positive integer or within a positive integer interval (including +∞). It can be expressed with a positive integer interval. For instance, under the `size` viewpoint, the `more-than-3` rooms class constrains the cardinal of the `rooms` field to be superior to 3 (i.e. in `]3 +inf[`).

Interface

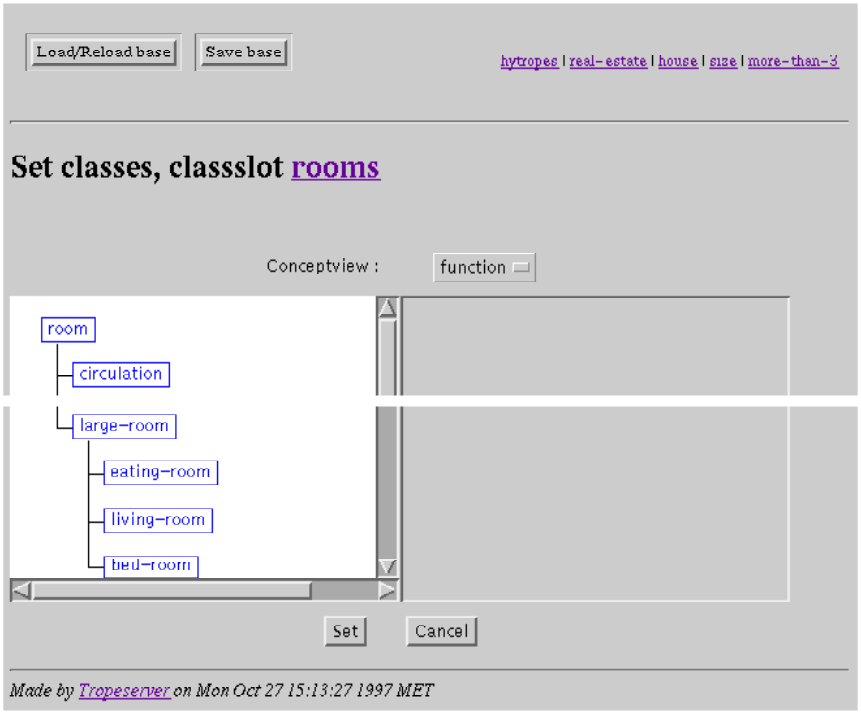


Screendump 18: Class field display.

Edit: ..... see Screendump 19, p. 48

Screendump 19: Class field edition.

View:..... see Screendump 18, p. 48
Set classes: ..... see Screendump 20, p. 49
Set exceptions: ..... see Screendump 22, p. 50
Set domains: ..... see Screendump 21, p. 49
Set intervals:..... see Screendump 23, p. 50
Set cardinal: ..... see Screendump 24, p. 50
Set default:..... see Screendump 40, p. 86
Set assigns:..... see Screendump 41, p. 86
Set filters:..... see Screendump 42, p. 86
Add methods:..... see Screendump 44, p. 87

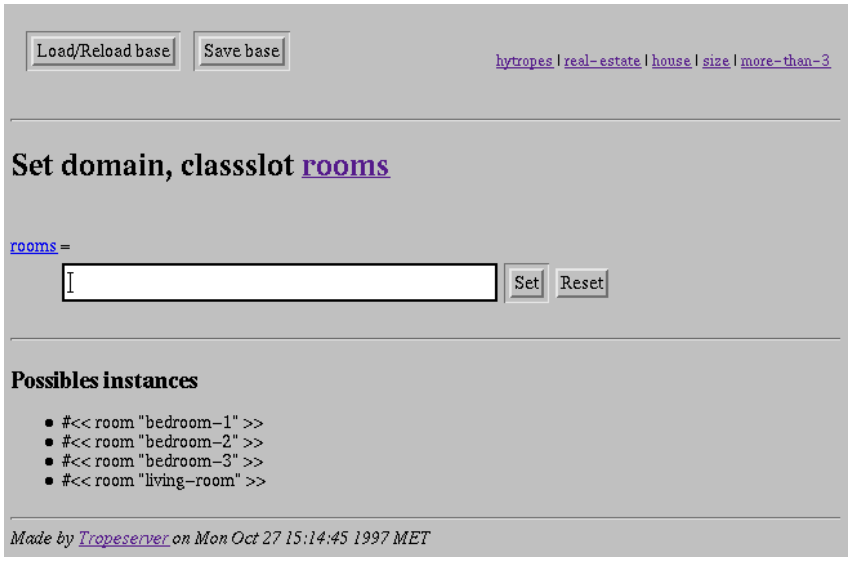


Screendump 20: Class field typing modification.

Set: ..... see Screendump 19, p. 48
Cancel:..... see Screendump 19, p. 48

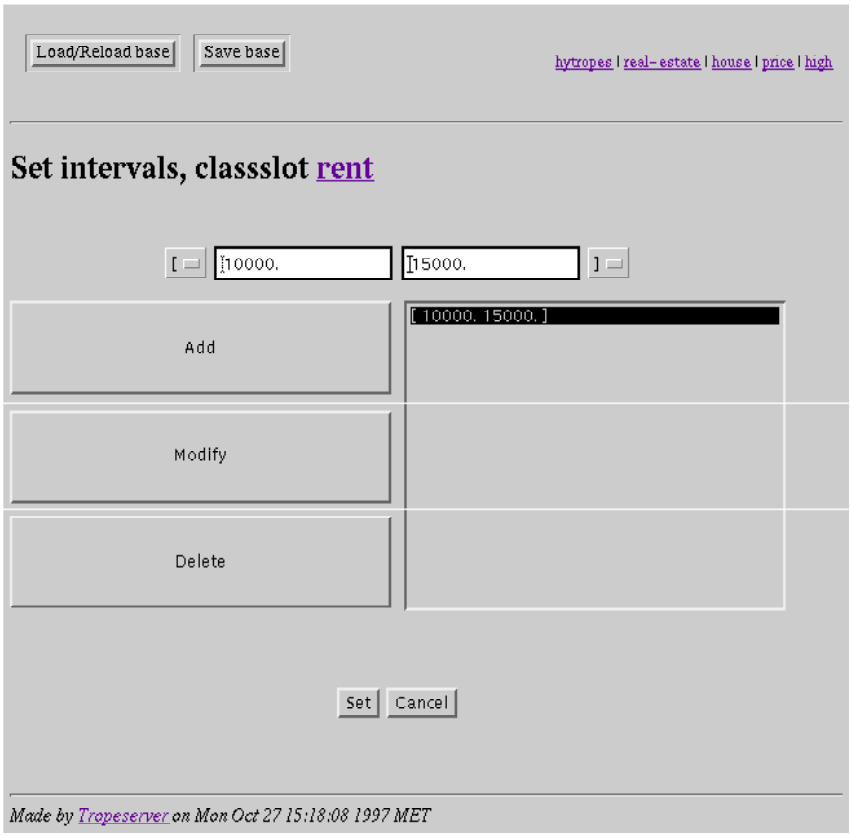
Screendump 21: Class field domain modification.

Set: ..... see Screendump 19, p. 48
Cancel:..... see Screendump 19, p. 48



Screendump 22: Class field except modification.

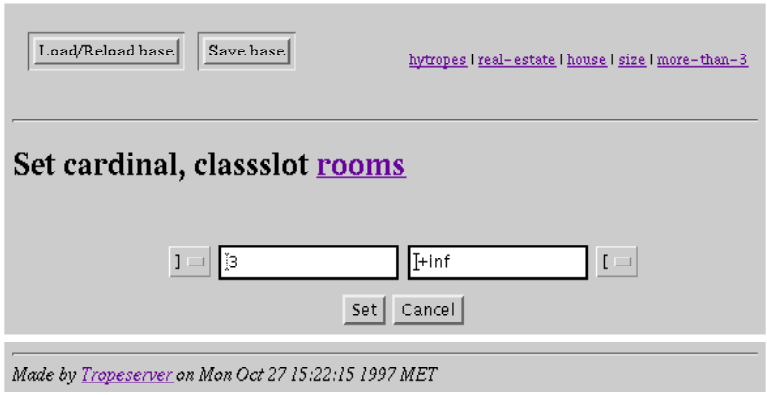
Set: ..... see Screendump 19, p. 48
Cancel:..... see Screendump 19, p. 48



Screendump 23: Class field intervals modification.

Set: ..... see Screendump 19, p. 48

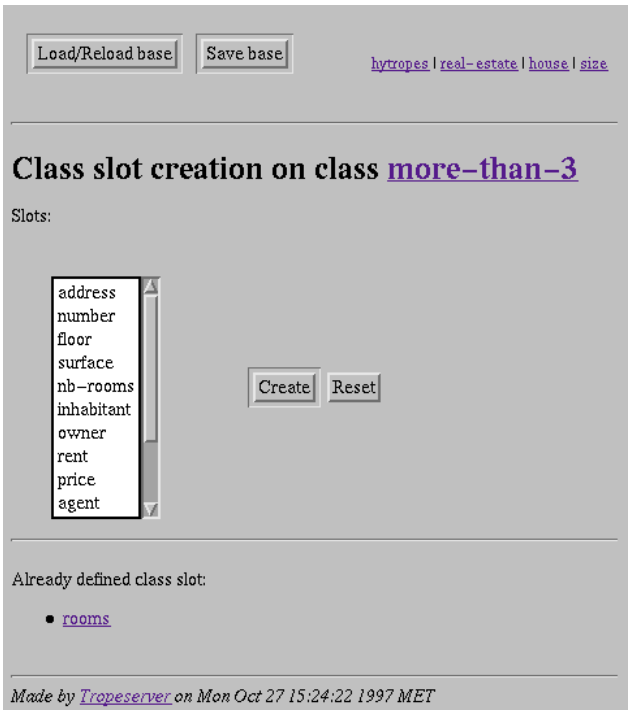
Cancel:..... see Screendump 19, p. 48



Screendump 24: Class field cardinality modification.

Set: ..... see Screendump 19, p. 48

Cancel:..... see Screendump 19, p. 48



Screendump 25: Class field creation.

Create: ..... see Screendump 19, p. 48
Reset: ..... see Screendump 25
<classslots>: ..... see Screendump 18, p. 48

API

(**tr-classslot-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a class field, `()` otherwise.

(**tr-create-classslot** *class classslotname*) function in [libtrp]

-> classslot, *class* $\text{class}_{\text{nac}}$, *classslotname* $\text{string}_{\text{nas, niks, adcs}}$

Creates and returns a new class field of name *classslotname* in the class *class*. The corresponding concept field is then visible under the concept viewpoint of the class. It will therefore be known of the system until deletion or initialisation.

(**tr-find-classslot** *class classslotname*) function in [libtrp]

->classslot, *class* $\text{class}_{\text{nac}}$, *classslotname* $\text{string}_{\text{nas, niks}}$

Retrieves and returns the class field of name *classslotname* in the class *class*. When no such class field is known, returns `()`.

(**tr-classslot** *class classslotname*) macro in [libtrp]

-> classslot, *class* $\text{class}_{\text{nac}}$, *classslotname* $\text{string}_{\text{nas, niks}}$

Retrieves and returns the class field of name *classslotname* in the class *class*. When no such class field is known, creates and returns it.

(**tr-delete-classslot** *classslot*) function in [libtrp]

-> boolean, *classslot* $\text{classslot}_{\text{nacs, neo}}$

Deletes the class field *classslot* from the environment. If there is no class field corresponding to the same concept field as *classslot* in the viewpoint, then the concept field is not visible

anymore under the concept viewpoint. Afterwards, the class field does not exist anymore in the environment.

(**tr-erase-classslot** *classslot*) function in [libtrp]
-> boolean, *classslot* classslot$_{nacs, neo}$
Deletes the class field *classslot* from the environment. Afterwards, the class field does not exist anymore in the environment.

(**tr-classslot-name** *classslot*) function in [libtrp]
-> string, *classslot* classslot$_{nacs}$
Returns the name of the class field *classslot*.

(**tr-classslot-class** *classslot*) function in [libtrp]
-> class, *classslot* classslot$_{nacs}$
Returns the class of the class field *classslot*.

(**tr-classslot-conceptslot** *classslot*) function in [libtrp]
-> conceptslot, *classslot* classslot$_{nacs}$
Returns the concept field corresponding to the class field *classslot*.

(**tr-get-classslot-type** *classslot*) function in [libtrp]
-> { class* }, *classslot* classslot$_{nacs}$
Returns the type of the class field *classslot*, `()` if it is not refined in the class field.

(**tr-set-classslot-type** *classslot type*) function in [libtrp]
-> boolean, *classslot* classslot$_{nacs}$, *type* { class$_{ivt}$* }$_{ivt, adf}$
Refines the type of the class field *classslot* by setting it to the new type *type*.

(**tr-get-classslot-domain** *classslot*) function in [libtrp]
-> { entity* }, *classslot* classslot$_{nacs}$
Returns the domain of the class field *classslot*, `()` if it is not refined in the class field.

(**tr-set-classslot-domain** *classslot domain*) function in [libtrp]
-> boolean, *classslot* classslot$_{nacs}$, *domain* { entity* }$_{ivd, adf}$
Refines the domain of the class field *classslot* by restricting it to the new domain *domain*.

(**tr-get-classslot-except** *classslot*) function in [libtrp]
-> { entity* }, *classslot* classslot$_{nacs}$
Returns the exceptions of the class field *classslot*, `()` if they are not refined in the class field.

(**tr-set-classslot-except** *classslot domain*) function in [libtrp]
-> boolean, *classslot* classslot$_{nacs}$, *domain* { entity* }$_{ive, adf}$
Refines the exceptions of the class field *classslot* by retracting the new domain *domain*.

(**tr-get-classslot-intervals** *classslot*) function in [libtrp]
-> { interval* }, *classslot* classslot$_{nacs}$
Returns the intervals of the class field *classslot*, `()` if they are not refined in the class field.

(**tr-set-classslot-intervals** *classslot intervals*) function in [libtrp]
-> boolean, *classslot* classslot$_{nacs}$, *intervals* (interval*)$_{ivi, adf}$
Refines the intervals of the class field *classslot* by restricting them to the new intervals *intervals*.

(**tr-get-classslot-card** *classslot*) function in [libtrp]

-> interval, *classslot* classslot$_{nacs}$

Returns the cardinal of the class field *classslot*, `()` if it is not refined in the class field.

(**tr-set-classslot-card** *classslot interval*) function in [libtrp]

-> boolean, *classslot* classslot$_{nacs,nct}$, *interval* interval$_{ivcd,\ adf}$

Refines the cardinal interval of the class field *classslot* by restricting it to the new interval *interval*.

Examples

```
? (tr-classslot-p
    (tr-create-classslot
      (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent") "comission"))
=  (tr-classslot  (tr-class  (tr-conceptview  (tr-concept  "agent")  "buyer")  "agent")
"comission")
? (tr-find-classslot
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent") "comission")
=  (tr-classslot  (tr-class  (tr-conceptview  (tr-concept  "agent")  "buyer")  "agent")
"comission")
? (tr-create-classslot
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
    "houses")
= (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent") "houses")
? (tr-classslot-name
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "comission"))
= "comission"
? (tr-classslot-class
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "comission"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
? (tr-get-classslot-intervals
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "comission"))
= #s(<undef>)
? (tr-set-classslot-intervals
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "comission")
    (list (tr-make-interval 0. t 100. t)))
= ((tr-make-interval 0. t 100. t))
? (tr-get-classslot-intervals
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "comission"))
= ((tr-make-interval 0. t 100. t))
? (tr-set-classslot-intervals
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")
                  "comission")
    (list (tr-make-interval 0. t 7. t)))
= ((tr-make-interval 0. t 7. t))
? (tr-set-classslot-intervals
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
                  "comission")
    (list (tr-make-interval 7. () 100. t)))
= ((tr-make-interval 7. () 100. t))
? (tr-set-classslot-domain
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "area")
    (list "meylan" "grenoble" "echirolles" "crolles" "montbonnot"
          "eybens" "saint-martin d'heres" "fontaine" "sassenage"
          "sessinnet" "saint-egreve"))
= ("meylan" "grenoble" "echirolles" "crolles" "montbonnot" "eybens"
"saint-martin d'heres" "fontaine" "sassenage" "sessinnet" "saint-egreve")
? (tr-set-classslot-except
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
                  "area")
```

```
    (list "meylan" "grenoble" "crolles" "montbonnot"
          "saint-martin d'heres"))
= ("meylan" "grenoble" "crolles" "montbonnot" "saint-martin d'heres")
? (tr-get-classslot-except
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "valuable")
                  "area"))
= ("meylan" "grenoble" "crolles" "montbonnot" "saint-martin d'heres")
? (tr-set-classslot-domain
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "area")
    (list "meylan" "grenoble" "crolles" "montbonnot"
          "saint-martin d'heres"))
= ("meylan" "grenoble" "crolles" "montbonnot" "saint-martin d'heres")
? (tr-get-classslot-domain
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "area"))
= ("meylan" "grenoble" "crolles" "montbonnot" "saint-martin d'heres")
? (tr-set-classslot-type
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "houses")
    (list
      (tr-find-class (tr-find-conceptview (tr-find-concept "house") "size") "palace")))
= ((tr-class (tr-conceptview (tr-concept "house") "size") "palace"))
? (tr-get-classslot-type
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "houses"))
= ((tr-class (tr-conceptview (tr-concept "house") "size") "palace"))
? (tr-set-classslot-card
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "houses")
    (tr-make-interval 30 t '+inf ()))
= (tr-make-interval 30 t '+inf ())
? (tr-get-classslot-card
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
                  "houses"))
= (tr-make-interval 30 t '+inf ())
? (tr-delete-classslot
    (tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
                  "houses"))
= t
? (tr-class-slots
    (tr-find-class (tr-find-conceptview (tr-find-concept "agent") "buyer") "agent"))
= ((tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent") "area")
   (tr-classslot   (tr-class   (tr-conceptview   (tr-concept   "agent")   "buyer")   "agent")
"comission")
? (tr-class-sorted-slots
  (tr-find-class
      (tr-find-conceptview
        (tr-find-concept "agent")
   "buyer")
      "agent"))
= ((tr-classslot (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent") "area")
   (tr-classslot   (tr-class   (tr-conceptview   (tr-concept   "agent")   "buyer")   "agent")
"comission")
```

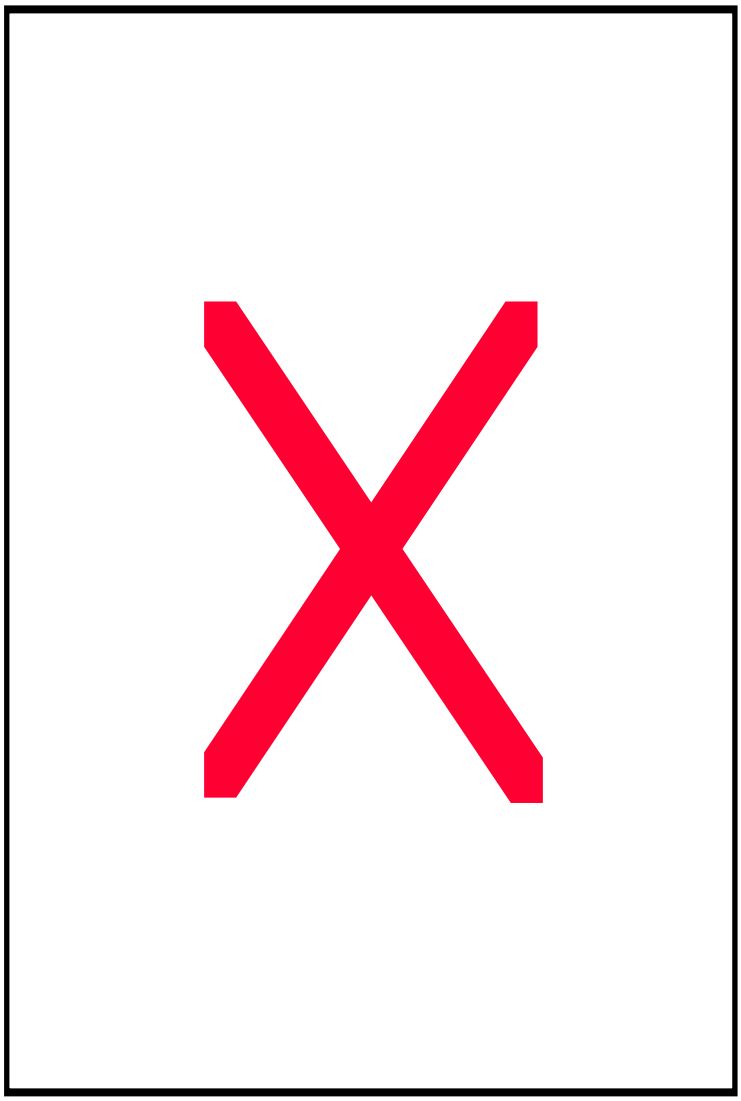
Filters

The structure of filters is the same as those of classes but they are not considered as classes. A filter is a definitional class description attached to a concept or to a set of classes (from different viewpoints). It aims at collecting the set of instances which satisfy its constraints. The filter is satisfied by any instance (of the concept or member of the above classes) which satisfies the constraints defined by the filter. They can be used for various tasks (related to instance retrieval). In particular, filters are used for specifying authorised field values (see the "Class field definition: data typing aspects/unary constraint" section) and inferring possible field values (see the "Computation by filter" section in the "Computing field values" part).

A filter is created through a special function and can be manipulated exactly as a class by adding constraints on it and retrieving its members through `tr-class-member`). However, filters do not have sub-classes (or sub-filters).

Note that filters are the only element which are not recoverable from their printed forms (i.e. cut and paste of a filter from the result of its creation will not return the filter).

Interface



Screendump 26: Filter creation, edition and execution.

<view>: ..... see Screendump 11, p. 34
<slot>:..... see Screendump 8, p. 29
Exec query ..... see Screendump 27, p. 56
Reset: ..... see Screendump 26

Screendump 27: Filter result.

<object>:..... see Screendump 31, p. 66

API

(**tr-filter-p** *anything*) function in [libtrp]

-> boolean, *anything* entity

Creates and returns a new filter from class *class* with no more constraint than the class *class*.

(**tr-create-filter** concept rest:*classes*) function in [libtrp]

-> filter, *concept* concept_{nak}, *classes* { class_{nac}* }$_{ivb}$

Creates and returns a new filter in concept concept from classes *classes* with no more constraint than those of the classes.

(**tr-delete-filter** *filter*) function in [libtrp]

-> boolean, *filter* $\text{filter}_{naft,\ neo}$

Deletes the filter *filter*. This inhibits it. This can only happen if the filter is as empty as at its creation: i.e. it has no classslots nor constraints. Afterwards, the filter does not exist anymore in the environment.

(**tr-erase-filter** *filter*) function in [libtrp]

-> boolean, *filter* $\text{filter}_{naft,\ neo}$

Deletes the filter *filter* and its classslots and constraints.

(**tr-filtered-objects** *filter*) function in [libtrp]

-> (object*), *filter* filter_{naft}

Returns the set of objects which satisfy the filter *filter*.

(**tr-concept-proper-filters** *concept*) function in [libtrp]

-> (filter*), *concept* concept_{nak}

Returns the set of filters of the concept *concept* which are not attached to any class.

(**tr-class-proper-filters** *class*) function in [libtrp]

-> (filter*), *class* class_{nac}

Returns the set of filters directly attached to the class *class*.

(**tr-class-filters** *class*) function in [libtrp]

-> (filter*), *class* class_{nac}

Returns the set of filters indirectly attached under the class *class*.

Examples

```
? (tr-class-proper-filters
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
= ()
? (setf my-filter
    (tr-create-filter
      (tr-concept "agent")
      (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")))
= (tr-filter (tr-concept "agent"))
? (tr-class-proper-filters
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
= ((tr-filter (tr-concept "agent")))
? (tr-class-proper-filters
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
= ()
? (tr-class-filters
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
= ((tr-filter (tr-concept "agent")))
? (tr-filter-p my-filter)
= (tr-filter (tr-concept "agent"))
? (tr-concept-proper-filter (tr-concept "agent"))
= ()
? (tr-class-filters
   (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
```

```
= ((tr-filter (tr-concept "agent")))
? (tr-filtered-objects my-filter)
= ()
? (setf my-other-filter (tr-create-filter (tr-concept "agent")))
= (tr-filter (tr-concept "agent"))
? (eq my-filter my-other-filter)
= ()
? (tr-concept-proper-filters (tr-concept "agent"))
= ((tr-filter (tr-concept "agent")))
```

Bridges

Viewpoints define class hierarchies which are not completely independent. As an example, in the `house` concept it is known that the objects which are members of the `residential` area under the `localisation` viewpoint and the `more-than-3` room class under the `size` viewpoint are also in the class of `high` priced houses under the `price` viewpoint. This is expressed with the help of bridges.

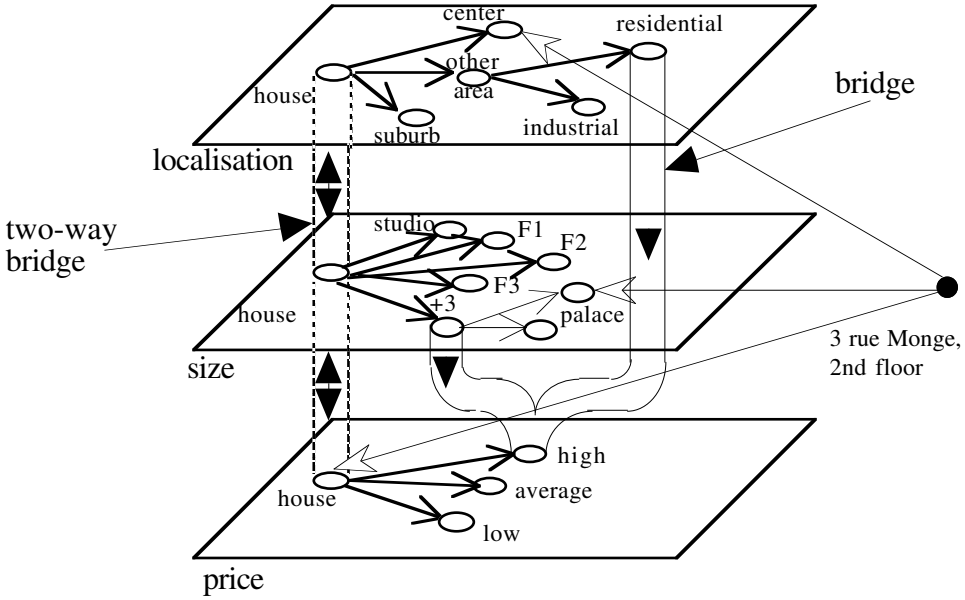


Figure 8: The `house` concept augmented with a bridge asserting that the houses located in a `residential` area with more than `three` rooms must have a `high` price. Other (bi-directional) bridges are those related the root classes of the viewpoints.

A bridge has a name and is made of a source which is a set of classes and a destination which is a class in another viewpoint. Thus, unlike a real bridge, a TROEPS bridge is oriented. It asserts that each member of all the source classes is also a member of the destination class.

This entails some consequences among which:

- The (combination of the) field restrictions of the sources classes can be inferred as having more (or the same) restrictions than the destination class. Installing bridges between classes does not lead to any consequence on the visibility of fields. However, it has consequences on how the field values (even hidden) are constrained. For instance, if any residential house must be high priced, this means that a particular house is not subject to be classified in the residential class when it is low priced, and this, even if the price field is not visible under the localisation viewpoint (see the "classification" section for more details about this).
- The source classes must all be from different viewpoints (otherwise, the bridge is either redundant or contradictory). The destination class must be from a different viewpoint than the viewpoints of source classes (otherwise this would result either in asserting that some classes are empty or some other — related by specialisation — are the same by definition).

Note that, conceptually, the root class of each viewpoint is related by a bridge with any other root class of any other viewpoint (represented as bi-directional bridges on the picture).

However, these implicit bridges are not physically represented in TROEPS. So, it is not possible to find them through the bridge related functions of concepts and classes.

Further constraints must be satisfied when installing bridges: a bridge must not lead to a contradiction in the inclusion relations induced by specialisation. As a matter of fact, some points are deduced by TROEPS:

- Each pair of bridges with the same source can only have, as destinations, classes related by specialisation.
- When considering a first viewpoint with a class *A* and two subclasses *A'* and *A"*, a second viewpoint with a class *B* and two sub-classes *B'* and *B"* plus two one-way bridges from *B'* to *A* and from *A'* to *B*, it can be deduced that (1) *A"* and *B"* are empty (2) *A*, *A'*, *B*, *B'*are the same class.
- Concerning circularity detection among bridges, the system detects and refuses only one kind of circular bridge sets: those which constrain any object triggering the bridge to migrate in a class non comparable with one of the sources of the bridge (see Figure 9).

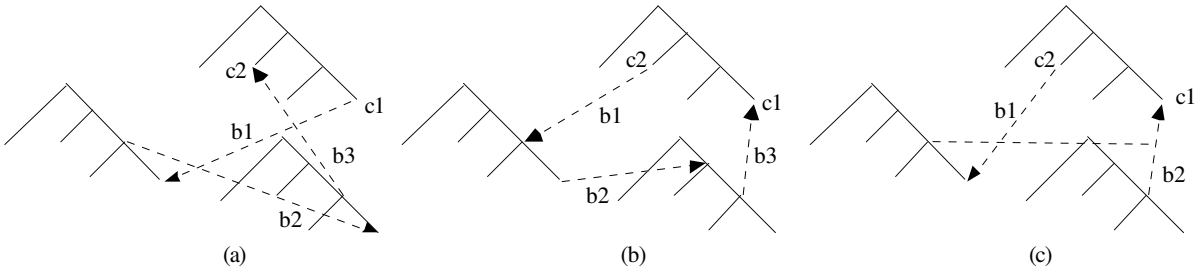


Figure 9: Examples of a circularity prohibited by TROEPS (a) and two others (b and c) non prohibited.

Finally, TROEPS is able to detect subsumption among bridges and to warn the user of this. The semantics of bridge subsumption is that a bridge is subsumed by another if and only if any possible object falling under it would also fall under the other and their destination classes are in specialisation relation. More formally, a bridge is subsumed by another if and only if for each class in the source of the latter there exists a super-class of it in that of the former and the destination of the former is a super-class of that of the latter. As an example, Figure 10 illustrates the fact that B1 is subsumed by B2 and B3 but B2 and B3 are non comparable by bridge subsumption since if all objects which fall under B3 also fall under B2, B3 is more precise than B2 (it leads to a more specific class).

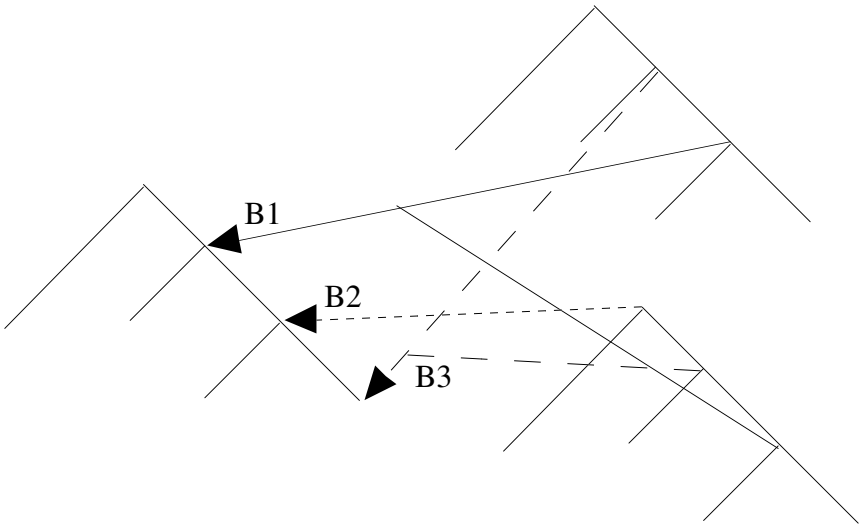
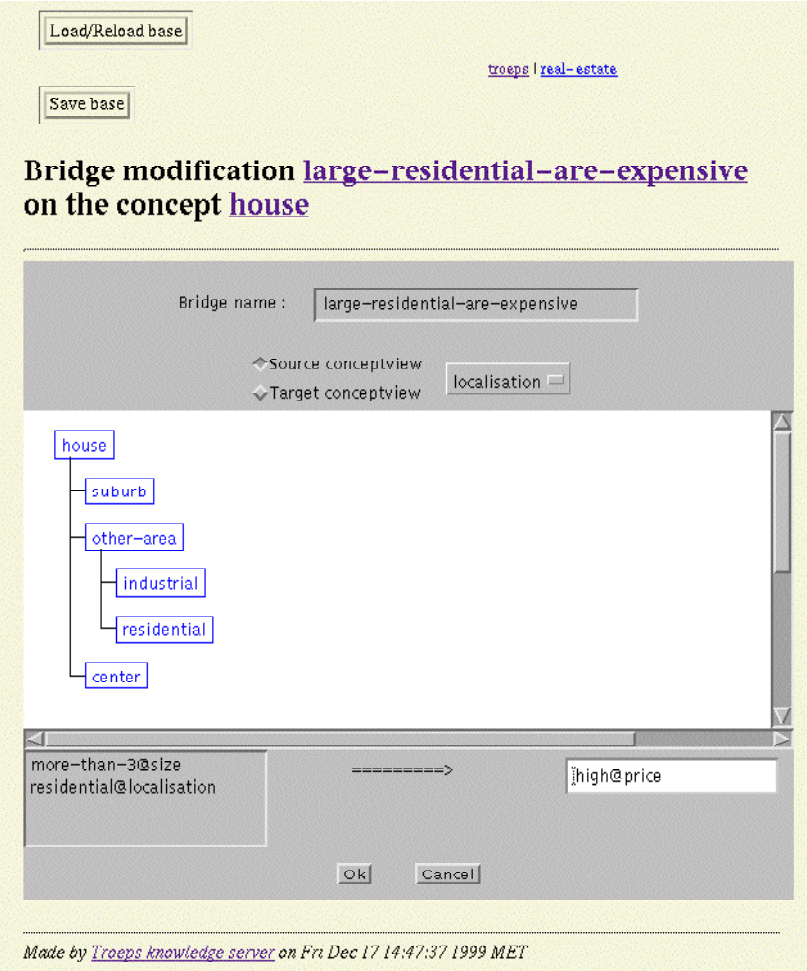


Figure 10: Bridge subsumption example: B1 is subsumed by both B2 and B3.

Interface

Screendump 28: Bridge display.
Edit: ..... see Screendump 29, p. 62
<class>: ..... see Screendump 14, p. 40

Screendump 29: Bridge edition.
View:..... see Screendump 28, p. 61
Modify:..... see Screendump 30, p. 62
<class>:..... see Screendump 15, p. 41



Screendump 30: Bridge creation and modification.
Ok:..... see Screendump 6, p. 23
Cancel:..... see Screendump 6, p. 23

API

(**tr-bridge-p** *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a bridge, `()` otherwise.

(**tr-create-bridge** *concept name source destination*) function in [libtrp]

-> bridge, *concept* concept_{nak}, *name* $\text{string}_{nas,adb,}$, *source* { class_{nac}* }$_{ivbp}$, *destination* class_{nac}

Creates and returns a new bridge between the source classes *source* and the destination class *destination*. This bridge is automatically inserted in the list of bridges of the concept and the lists of bridges of the concerned classes. The consequences of the bridge establishment on the types of classes under the different viewpoints are computed and notified.

Error messages: bik, rbb, rba

(**tr-find-bridge** *concept bridgename*) function in [libtrp]

->bridge, *concept* concept_{nak}, *bridgename* string_{nas}

Retrieves and returns the bridge of name *bridgename* in the concept *concept*. When no such bridge is known, returns `()`.

(**tr-bridge** *concept bridgename source destination*) function in [libtrp]

->bridge, *concept* $\text{concept}_{\text{nak}}$, *bridgename* $\text{string}_{\text{nas}}$, *source* { $\text{class}_{\text{nac}}$* }$_{\text{ivbp}}$, *destination* $\text{class}_{\text{nac}}$

Retrieves and returns the bridge of name *bridgename* in the concept *concept*. When no such bridge is known, creates and returns it.

(**tr-delete-bridge** *bridge*) function in [libtrp]

-> boolean, *bridge* $\text{bridge}_{\text{nap, neo}}$

Deletes the bridge *bridge* from its concept. The bridge is automatically removed from the list of bridges of the concept and the list of bridges of source and destination classes.

(**tr-erase-bridge** *bridge*) function in [libtrp]

-> boolean, *bridge* $\text{bridge}_{\text{nap, neo}}$

Deletes the bridge *bridge* from its concept. The bridge is automatically removed from the list of bridges of the concept and the lists of bridges of source and destination classes.

(**tr-bridge-concept** *bridge*) function in [libtrp]

-> concept, *bridge* $\text{bridge}_{\text{nap}}$

Returns the concept of the bridge *bridge*.

(**tr-bridge-name** *bridge*) function in [libtrp]

-> string, *bridge* $\text{bridge}_{\text{nap}}$

Returns the name of the bridge *bridge*.

(**tr-bridge-rename** *bridge name*) function in [libtrp]

-> string, *bridge* $\text{bridge}_{\text{nap}}$, *name* $\text{string}_{\text{nas, adp}}$

Changes the name of the bridge *bridge* for the string *name*.

(**tr-bridge-source** *bridge*) function in [libtrp]

-> { class* }, *bridge* $\text{bridge}_{\text{nap}}$

Returns the set of source classes of the bridge *bridge*.

(**tr-bridge-destination** *bridge*) function in [libtrp]

-> class, *bridge* $\text{bridge}_{\text{nap}}$

Returns the destination class of the bridge *bridge*.

Examples

```
? (tr-bridge-p ())
= ()
? (tr-bridge-p
    (tr-create-bridge
      (tr-concept "agent") "bad-agents-are-preferable"
      (list (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
      (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")))
= (tr-bridge (tr-concept "agent") "bad-agents-are-preferable")
? (tr-class-bridges-in (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad"))
= ((tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
? (tr-class-bridges-out
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
= ((tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
? (tr-concept-bridges (tr-concept "agent"))
= ((tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
? (tr-bridge-concept (tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
```

```
= (tr-concept "agent")
? (tr-bridge-name (tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
= "bad-agents-are-preferable"
? (tr-bridge-source (tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
= ((tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
? (tr-bridge-destination (tr-bridge (tr-concept "agent") "bad-agents-are-preferable"))
= (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")
```

Objects

An object is an instance of a concept. It is made of a collection of field values corresponding to fields defined in the concept. Fields, but key fields, can be unfilled (i.e. have no value). For a value of the key there exists only one corresponding object. This ensure the identity and the identifiability of the object. An object may evolve during its life-cycle (from its creation to its deletion) but the key field values cannot change.

As an example, in the real estate base, a particular house is represented by an instance of the house concept whose `address` is `"rue Monge"`, `number` is `3`, `floor` is `2`, `area` is `"Latin quarter"` (I know you knew), `rent-price` is `4500`. (Yeah, these are French francs) and `nb-rooms` is `2` (so, you cannot come with your kids). It is noteworthy that TROEPS is suited to real estate purpose since it is possible to have unfilled fields such as the one concerning the `surface` (floor dimension) of the apartment. Since the key of a house instance is the `address` plus a `floor` plus a `number`, this house is uniquely identified and that is why the mail can reach it (however, for mail concerning TROEPS, you can send it to the address in the introductory note, thanks).

An instance is attached to one and only one class, for each viewpoint. This is described by the `is-in` field, the value of which is the set of classes of the object (when there is no class for a viewpoint, it is assumed that the value is the root class of that viewpoint). This object is attached to the `average` (for Paris of course) class under the `price` viewpoint, the `center` for the `localisation` viewpoint and the `F2` for the `size` viewpoint. The instance belongs to its initial class and to any of its super-classes, under each viewpoint. Of course, all the field values of the above instance (the one whose `address` is `"rue Monge"`, `number` is `3` and whose `floor` is `2`) satisfy the constraints put on them by the different classes. Moreover, even the unknown fields must satisfy the constraints (so, it is considered that the real value of the field satisfies them). This is asserted by the fact that the user (or one of his programs) has (voluntarily) attached the object to the class. For instance, the real estate agent lets the customer think that there is a bridge from a flat whose `rent-price` is higher than `4000` Francs and a flat whose `surface` is more than `40` m^2. Such a bridge would enable to deduce that the `size` of the above flat is more than `40` m^2. However, there is no such bridge in the base (read your contract again).

A complete instance, when considered from a global viewpoint, shows its complete information, i.e. values for each field of the concept. On the other hand, when considered under a specific viewpoint, only fields known under this viewpoint can be seen, so a viewpoint provides a partial view on the instance (any other field is hidden under the viewpoint). As an example, the object representing the house described above has an `address`, door `number`, `floor` number under each viewpoint plus:

- a number of `room` under the `size` viewpoint;
- a `rent-price` under the `price` viewpoint (yeah, the extra charge is also unfilled);
- an `area` under the `localisation` viewpoint.

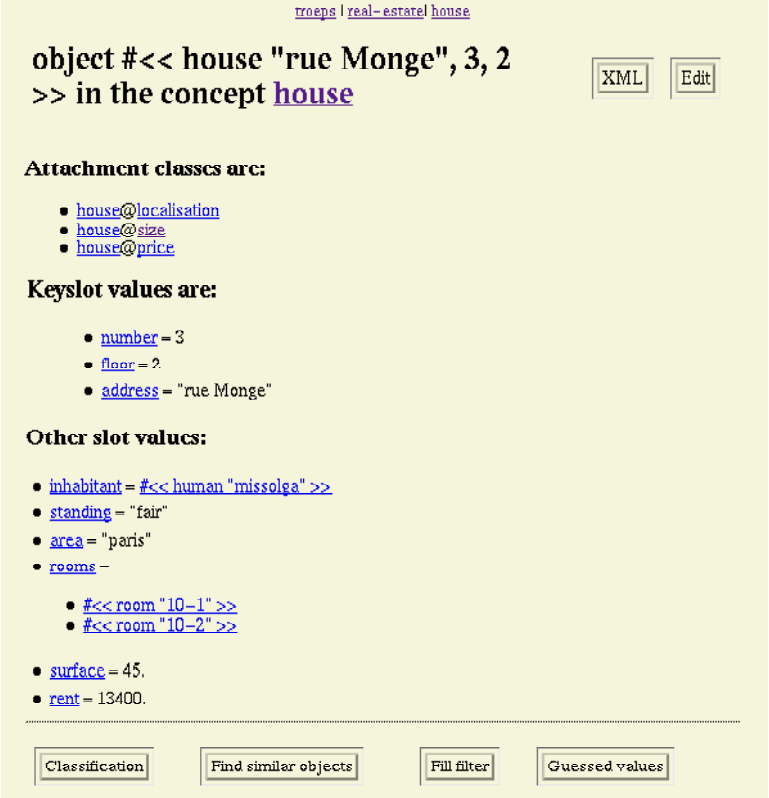
Class definition, attachment and values

There is a subtle game between class definition, attachment and field values. As a matter of fact, any change of one of them must comply with the definition of the others: changing attachment can only be done if the slot values comply with the new class definition, changing a slot value can only be achieved if the value complies with the definition of the current

attachment classes and changing a class definition requires that the attached objects complies the new definition.

In the current implementation, such an operation can only be achieved if it does not conflict with the others. This also applies to the knowledge bases in textual format.

Interface

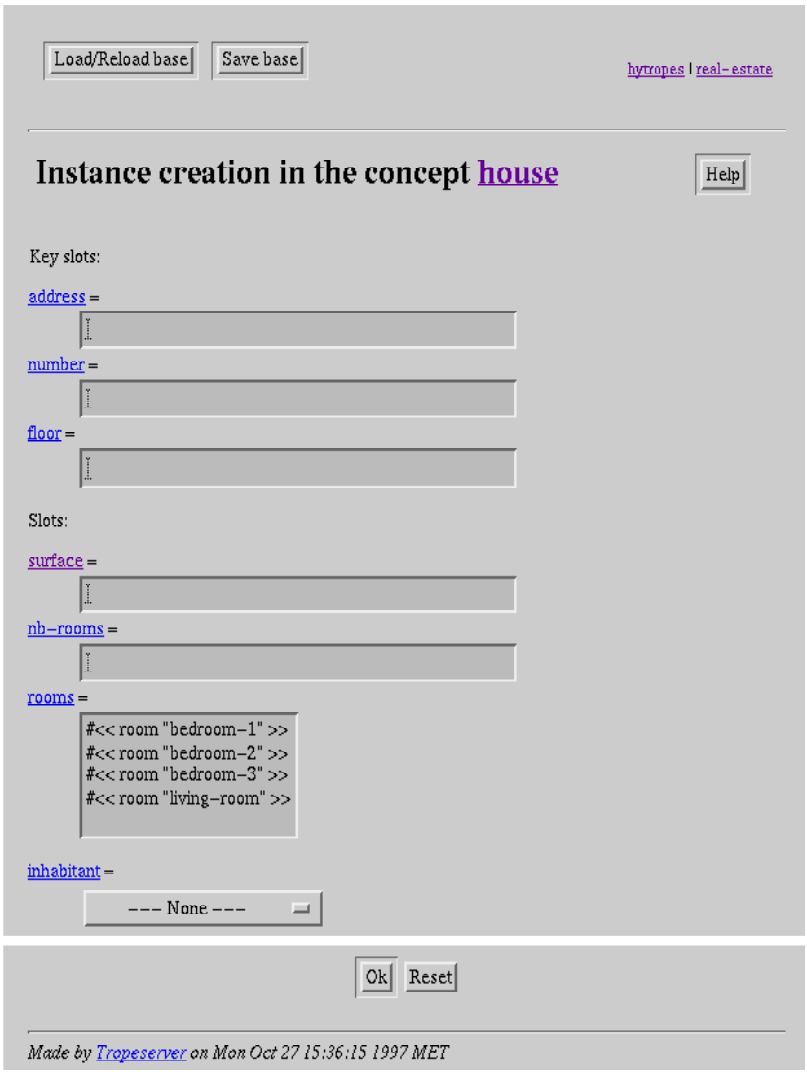


Screendump 31: Object display.

Edit: ..... see Screendump 32, p. 67
<class>: ..... see Screendump 14, p. 40
<view>: ..... see Screendump 11, p. 34
<slot>: ..... see Screendump 8, p. 29
<object>: ..... see Screendump 31
Classification: ..... see Screendump 46, p. 98
Find similar instances: ..... see Screendump 49, p. 104
Fill filter (this is inspired from Ontosaurus): ..... see Screendump 26, p. 56
Guess values: ..... see Screendump 45, p. 88

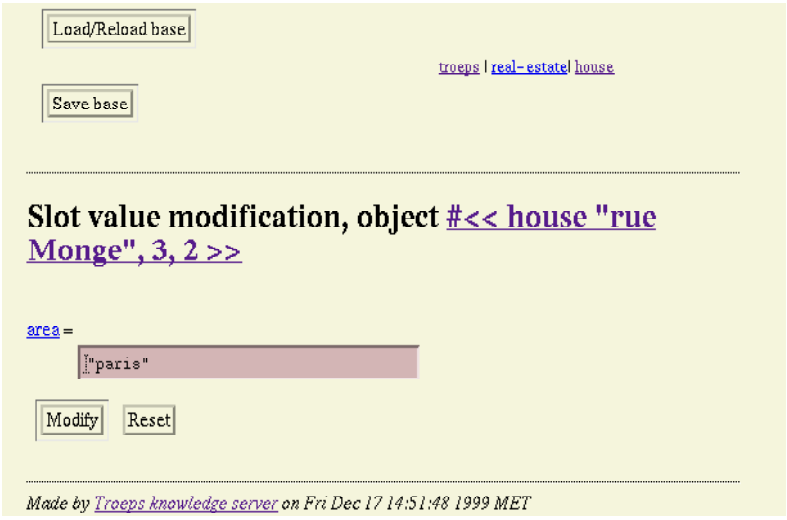
Screendump 32: Object edition. The key field values are not editable.

View:..... see Screendump 31, p. 66
Attach class:..... see Screendump 35, p. 69
Modify:..... see Screendump 33, p. 68
<class>:..... see Screendump 15, p. 41
<view>:..... see Screendump 12, p. 35
<slot>:..... see Screendump 9, p. 30
Modify (slot):..... see Screendump 34, p. 68
<object>:..... see Screendump 32
Classification:..... see Screendump 46, p. 98
Find similar instances:..... see Screendump 49, p. 104
Fill filter (this is inspired from Ontosaurus):..... see Screendump 26, p. 56
Guess values:..... see Screendump 45, p. 88



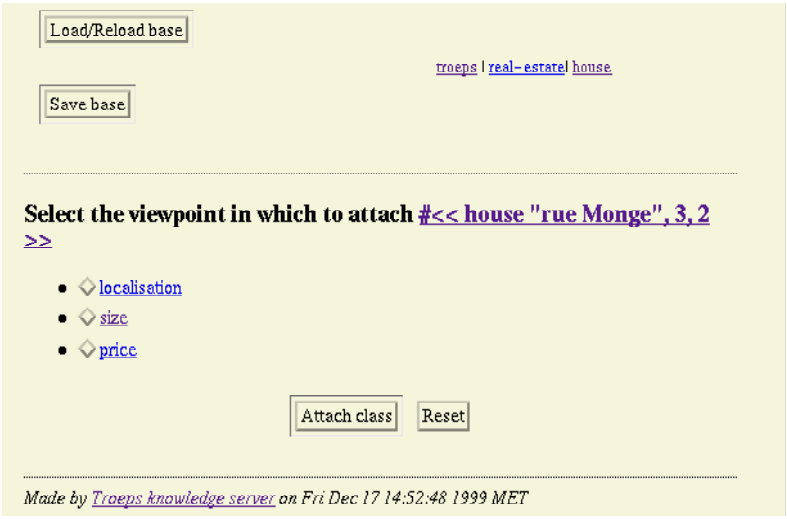
Screendump 33: Object creation and modification.

Ok:..... see Screendump 32, p 67
Reset: ..... see Screendump 33



Screendump 34: Field value modification.

Modify:..... see Screendump 32, p 67
Reset: ..... see Screendump 34



Screendump 35: Object attachment.

<view> ..... see Screendump 11, p. 34
Attach class:..... see Screendump 47, p 98
Reset: ..... see Screendump 35

API

(tr-object-p *anything*) predicate in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is an object, `()` otherwise.

(tr-create-object *concept keyval*) function in [libtrp]

-> object, *concept* $\text{concept}_{\text{nak}}$, *keyval* (entity*)$_{\text{ivk, aeo}}$

Creates and returns a new instance of the concept *concept* with the key specified by the list of key field values *keyval*. It will therefore be known of the system until deletion or initialisation.

(tr-create-complete-object *concept classlist slotvallist*) function in [libtrp]

-> object, *concept* $\text{concept}_{\text{nak}}$, *classlist* { $\text{class}_{\text{nac}}$* }$_{\text{ivps}}$, *slotvallist* { $(\text{string}_{\text{nas,nsa}}$. entity$)_{\text{btv}}$* }$_{\text{ivk, aeo}}$

Creates and returns a new instance of the concept *concept* with the list of field values given in *slotvallist* (which must at least contain the values for key slots). The created object is attached to the classes in *classlist*. It will be known of the system until deletion or initialisation.

(tr-find-object *concept keyval*) function in [libtrp]

-> object, *concept* $\text{concept}_{\text{nak}}$, *keyval* (entity*)$_{\text{ivk}}$

Retrieves and returns the instance of concept *concept* with key *keyval*. When no such object is known, returns `()`.

(tr-object *concept keyval*) macro in [libtrp]

-> object, *concept* $\text{concept}_{\text{nak}}$, *keyval* (entity*)$_{\text{ivk}}$

Retrieves and returns the instance of concept *concept* with key *keyval*. When no such object is known, creates and returns it.

(tr-delete-object *object*) function in [libtrp]

-> boolean, *object* $\text{object}_{\text{nao, neo}}$

Deletes the object *object* from the environment. Afterwards, the object does not exist anymore in the environment.

(**tr-erase-object** *object*) function in [libtrp]

-> boolean, *object* object$_{nao, neo}$

Deletes the object *object* from the environment. Afterwards, the object does not exist anymore in the environment.

(**tr-object-key** *object*) function in [libtrp]

-> (entity*), *object* object$_{nao}$

Returns the list of key field values of the object *object*.

(**tr-object-concept** *object*) function in [libtrp]

(**tr-is-a** *object*) macro in [libtrp]

-> concept, *object* object$_{nao}$

Returns the concept of the object *object*.

(**tr-is-in** *object rest: views*) function in [libtrp]

-> { class* }, *object* object$_{nao}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Returns the set of classes to which the object *object* is attached. If the optional set of concept views *views* is provided, only the classes in the concerned views are returned.

(**tr-is-in-under** *object conceptview*) function in [libtrp]

-> class, *object* object$_{nao}$, *conceptview* conceptview$_{nakv}$

Returns the class to which the object *object* is attached under the concept viewpoint *conceptview*.

(**tr-attach** *object class*) function in [libtrp]

-> class, *object* object$_{nao}$, *class* class$_{nac, nik}$

Attaches the object *object* to the class *class*. The previous attachment class of the object under the same concept viewpoint is no longer valid. Note that the compliance of the object with the class definition is checked before attachment; in case of failure, nothing is changed.

(**tr-set-value** *object slotname value rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nas, ukks}$,*value* entity$_{ivsv, cmk, kcv, ccv}$,
views { conceptview$_{nakv}$* }$_{ivsp}$

Sets the value of the field of name *slotname* of the object *object* to the value *value*. If the value in the field is any other value, an error is raised. The compliance of the new value *value* with the type of the classslot in each class to which the object is attached is checked. If it fails, nothing is changed and the error `ivsv` is raised. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-unset-value** *object slotname rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nas, ukks}$,*views* { conceptview$_{nakv}$* }$_{ivsp}$

Resets the value of the field of name *slotname* of the object *object*. The old value of the field, if any, is discarded. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-get-value** *object slotname rest: views*) function in [libtrp]

-> value, *object* object$_{nao}$, *slotname* string$_{nas, ukks}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Returns the value of the field of name *slotname* of the object *object* to the value *value*. If no value has been provided for the field, the `(unbound-slot-value)` value is returned. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-modify-value** *object slotname value rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nas, ukks}$,*value* entity$_{ivsv, cmk, kcv, ccv}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Sets the value of the field of name *slotname* of the object *object* to the value *value*. If the value in the field is any other value, no error is raised. The compliance of the new value *value* with the type of the classslot in each class to which the object is attached is checked. If it fails, nothing is changed and the error `ivsv` is raised. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-add-value** *object slotname value rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nas, ukks,ncs}$,*value* entity$_{ivsv, cmk, kcv, ccv}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Adds the value *value* to the value of the field of name *slotname* of the object *object*. The compliance of the value with the type of the field is checked. If it fails, nothing is changed. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-rem-value** *object slotname value rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nsa, ukks,ncs}$,*value* entity$_{ivsv, cmk, kcv, ccv}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Removes the value *value* from the field of name *slotname* of the object *object*. The compliance of the value with the type of the field is checked. If it fails, nothing is changed. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

(**tr-mem-value** *object slotname value rest: views*) function in [libtrp]

-> boolean, *object* object$_{nao}$, *slotname* string$_{nsa, ukks,ncs}$,*value* entity$_{ivsv}$, *views* { conceptview$_{nakv}$* }$_{ivsp}$

Returns the value *value* if it is a member of the value of the field of name *slotname* of the object *object* and `()` if it is not. If the optional set of concept views *views* is provided, the action only happens if the field is visible under at least one of the views.

Examples

```
? (tr-object-p ())
= ()
? (tr-object-p (tr-create-object (tr-concept "agent") '("immo38")))
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-find-object (tr-concept "agent") '("immo38"))
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-create-complete-object
    (tr-concept "agent")
    (list (tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
          (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad"))
   '(("name" . "first-immo")
     ("comission" . 6.5)
     ("area" . "grenoble")))
```

```
= (tr-object (tr-concept "agent") '("first-immo"))
? (tr-concept-instances (tr-concept "agent"))
= ((tr-object (tr-concept "agent") '("immo38"))
   (tr-object (tr-concept "agent") '("first-immo")))
? (tr-class-attached (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
= ((tr-object (tr-concept "agent") '("immo38")))
? (tr-class-attached (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad"))
= ()
? (tr-class-members (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
= ((tr-object (tr-concept "agent") '("immo38"))
   (tr-object (tr-concept "agent") '("first-immo")))
? (tr-create-object (tr-concept "agent") '("bond, my name is bond"))
= (tr-object (tr-concept "agent") '("bond, my name is bond"))
? (tr-delete-object (tr-object (tr-concept "agent") '("bond, my name is bond")))
= t
? (tr-object-key (tr-find-object (tr-concept "agent") '("immo38")))
= ("immo38")
? (tr-object-concept (tr-find-object (tr-concept "agent") '("immo38")))
= (tr-concept "agent")
? (tr-is-in (tr-find-object (tr-concept "agent") '("immo38")))
= ((tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent")
   (tr-class (tr-conceptview (tr-concept "agent") "seller") "agent"))
? (tr-is-in (tr-find-object (tr-concept "agent") '("first-immo")))
= ((tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite")
   (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")
? (tr-is-in-under
    (tr-find-object (tr-concept "agent") '("first-immo"))
    (tr-conceptview (tr-concept "agent") "seller"))
= (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")
? (tr-attach
    (tr-find-object (tr-concept "agent") '("immo38"))
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
? (tr-class-attached
    (tr-class (tr-find-conceptview (tr-concept "agent") "buyer") "bad"))
= ((tr-object (tr-concept "agent") '("immo38")))
? (tr-is-in-under
    (tr-find-object (tr-concept "agent") '("immo38"))
    (tr-conceptview (tr-concept "agent") "buyer"))
= (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad")
? (tr-filtered-objects my-filter)
= ((tr-object (tr-concept "agent") '("first-immo")))
? (tr-set-classslot-domain
    (tr-create-classslot my-filter "area")
    (list "meylan" "montbonnot"))
= ("meylan" "montbonnot")
? (tr-filtered-objects my-filter)
= ()
? (tr-get-value (tr-object (tr-concept "agent") '("immo38")) "area")
= #s(<undef>)
? (tr-set-value (tr-object (tr-concept "agent") '("immo38")) "area" "saint-egreve")
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-get-value (tr-object (tr-concept "agent") '("immo38")) "area")
= "saint-egreve"
? (tr-modify-value (tr-object (tr-concept "agent") '("immo38")) "area" "grenoble")
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-get-value (tr-object (tr-concept "agent") '("immo38")) "area")
= "grenoble"
? (tr-unset-value (tr-object (tr-concept "agent") '("immo38")) "area")
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-get-value (tr-object (tr-concept "agent") '("immo38")) "houses")
= #s(<undef>)
? (tr-get-value (tr-object (tr-concept "agent") '("immo38")) "houses")
= #s(<undef>)
? (tr-set-value (tr-object (tr-concept "agent") '("immo38")) "houses"
    (list (tr-object (tr-concept "house") '("meylan" 176 3))))
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-add-value (tr-object (tr-concept "agent") '("immo38")) "houses"
    (tr-object (tr-concept "house") '("rue Monge" 3 2)))
```

```
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-mem-value
    (tr-object (tr-concept "agent") '("immo38"))
    "houses"
    (tr-object (tr-concept "house") '("rue Monge" 3 2)))
= ((tr-object (tr-concept "house") '("rue Monge" 3 2)))
? (tr-rem-value
    (tr-object (tr-concept "agent") '("immo38"))
    "houses"
    (tr-object (tr-concept "house") '("meylan" 176 3)))
= (tr-object (tr-concept "agent") '("immo38"))
? (tr-get-value
    (tr-object (tr-concept "agent") '("immo38"))
    "houses")
= ((tr-object (tr-concept "house") '("rue Monge" 3 2)))
```

Annotations

The current version of TROEPS contains a very rough mechanism for annotating fields and objects. Thus, any TROEPS entity and any object field can receive annotations. Users may declare labels for annotations (preventing the concurrent use of the same label), assign and retrieve annotations in fields objects. The current implementation does not provide any mechanism for securing annotations (setting an annotation erases any previous value for it). So, using annotations requires a great deal of programming. However, the annotation system, when unused, adds only one additional field per object and does not waste additional time but for allocating memory.

The set of already reserved annotation is presented in Appendix 9.

Typed annotations are saved in a file which extension is `.adf`. Others are not saved.

API

(**tr-declare-annotation** *label comment optional: type const*) function in [libtrp]

-> label, *label* string$_{nas,ada}$, *comment* string$_{nas}$, *type* string|concept$_{ivt}$, *const* string$_{nas,\ ivc}$

Declares the label *label* for annotating objects, raise an error if the label is already declared. If no type and constructor is given the annotation will simply not be saved.

(**tr-declared-annotation-p** *label*) function in [libtrp]

-> string|(), *label* string$_{nas}$

Checks if the label *label* is declared for annotations. If so, returns its comment otherwise returns `()`.

(**tr-set-slot-annotation** *object slotname label annotation*) function in [libtrp]

-> object, *object* object$_{nao}$, *slotname* string$_{nas,ukks}$, *label* string$_{nas,naa}$, *annotation* entity$_{,\ bta}$

Sets the annotation *annotation* under the label *label* for the field *slotname* of the object *object*. This primitive does not check whether there is already a value or not: it just replaces it.

(**tr-get-slot-annotation** *object slotname label*) function in [libtrp]

-> object, *object* object$_{nao}$, *slotname* string$_{nas,ukks}$, *label* string$_{nas,naa}$

Gets and returns the annotation under the label *label* for the field *slotname* of the object *object*. Returns `()` if no such annotation is defined.

(**tr-rem-slot-annotation** *object slotname label*) function in [libtrp]

-> object, *object* object$_{nao}$, *slotname* string$_{nas,ukks}$, *label* string$_{nas,naa}$

Erases the annotation under the label *label* for the field *slotname* of the object *object*.

The following primitives allow to annotate any of the TROEPS entities defined above (base, concepts, viewpoints, classes, instances, etc.).

(**tr-set-annotation** *entity label annotation*) function in [libtrp]

-> entity, *entity* entity$_{nao}$, *label* string$_{nas,naa}$, *annotation* entity$_{,\ bta}$

Sets the annotation *annotation* under the label *label* for the entity *entity*. This primitive does not check whether there is already a value or not: it just replaces it.

(**tr-get-annotation** *entity label*) function in [libtrp]

-> entity, *entity* entity$_{nao}$, *label* string$_{nas,naa}$

Gets and returns the annotation under the label *label* for the entity *entity*. Returns `()` if no such annotation is defined.

(**tr-rem-annotation** *entity label*) function in [libtrp]

-> entity, *entity* entity_{nao}, *label* $\text{string}_{nas,naa}$

Erases the annotation under the label *label* for the entity *entity*.

Examples

```
? (tr-declared-annotation-p "my-opinion")
= ()
? (tr-declare-annotation "my-opinion" "Here is my opinion on the considered item")
= "Here is my opinion on the considered item"
? (tr-declared-annotation-p "my-opinion")
= "Here is my opinion on the considered item"
? (tr-set-annotation
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good")
    "my-opinion"
    "Here I put good agents as far as I can judge")
= "Here I put good agents as far as I can judge"
? (tr-set-annotation
    (tr-object (tr-concept "agent") '("first-immo"))
    "my-opinion"
    "Not really the first, but at least, not the last")
= "Not really the first, but at least, not the last"
? (tr-get-annotation
   (tr-object (tr-concept "agent") '("first-immo"))
  "my-opinion")
= "Not really the first, but at lest, not the last")
? (tr-rem-annotation
   (tr-object (tr-concept "agent") '("first-immo"))
   "my-opinion")
= ()
? (tr-get-annotation
   (tr-object (tr-concept "agent") '("first-immo"))
   "my-opinion")
= ()
? (tr-set-slot-annotation
    (tr-object (tr-concept "agent") '("first-immo"))
    "area"
    "my-opinion"
    '("the agent lives in area" "saint-egreve"))
= ("the agent lives in area" "saint-egreve")
? (tr-get-slot-annotation
    (tr-object (tr-concept "agent") '("first-immo"))
    "area"
    "my-opinion")
= ("the agent lives in area" "saint-egreve")
? (tr-rem-slot-annotation
    (tr-object (tr-concept "agent") '("first-immo"))
    "area"
    "my-opinion")
= ()
? (tr-get-slot-annotation
    (tr-object (tr-concept "agent") '("first-immo"))
    "area"
    "my-opinion")
= ()
```

Lexicon

Facility for manipulating a lexicon has been added to the TROEPS system. The lexicon helps both the designer and the user of a knowledge base in providing the definitions of the terms used in the knowledge base (as identifiers) and relating these terms.

Description of lexicon elements

A lexicon is a simple set of lexitems. Each lexitem represents a particular term. It is characterised by the following information:

- its name which corresponds to the term itself (e.g., house);
- its definition which is a string defining the term (e.g. a building in which people, usually one family, live in);
- its hyperonym which is a lexitem representing a term more general than the current one (e.g. building);
- its hyponyms which are lexitems representing terms more specific than the current one (e.g. semi-detached house, cottage);
- its synonyms which are other lexitem representing terms with the same meaning (e.g. apartment for flat);
- its antonyms which are other lexitem representing terms with opposite meanings (e.g. non furnished for furnished);
- its bibliography which is set of bibliographic references (unspecified) related to the term (e.g. the place where the definition has been found: "Cambridge international dictionary of English, Cambridge university press, Cambridge (UK), 1995");
- its related objects which are TROEPS entities implementing in one sense or another the term (e.g. the `house` concept and all the root classes of this concept).

The relationship between these components of lexitems are managed correctly, for instance, if furnished is declared as an antonym of non furnished then non furnished is an antonym of furnished (however, if symmetry is dealt with correctly, this is not the case for transitivity neither for synonyms nor hyponyms).

Linking lexicon elements to TROEPS objects

Lexicons can be saved in files and loaded at any time. They can also be linked to the entities of a knowledge base. This means that for any entity, if its name is a term corresponding to a lexitem, then the entity will be added to the related objects of the lexitem and the lexitem will be considered as the lexical equivalent of the entity (there can be several entities related to a lexitem while there is only one lexitem connected with an entity).

A knowledge base can have one or several lexicons attached to it. This means that at knowledge base load time, the lexicon will be loaded and the knowledge base will be indexed by the terms in the lexicons.

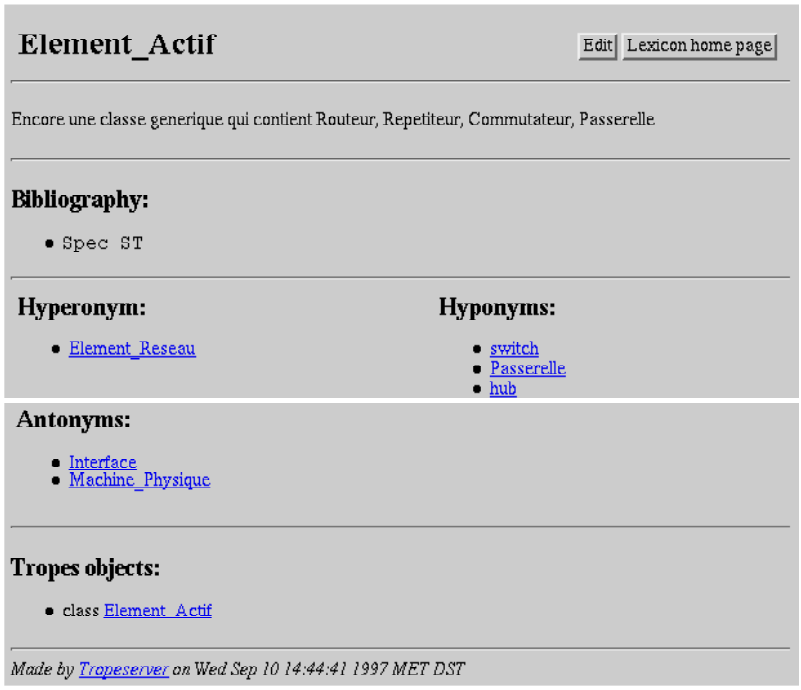
Interface

Screendump 36: Lexicon home page.

OK: ..... see Screendump 38, p. 78

View complete lexicon:

Create lexitem: ..... see Screendump 39, p. 79

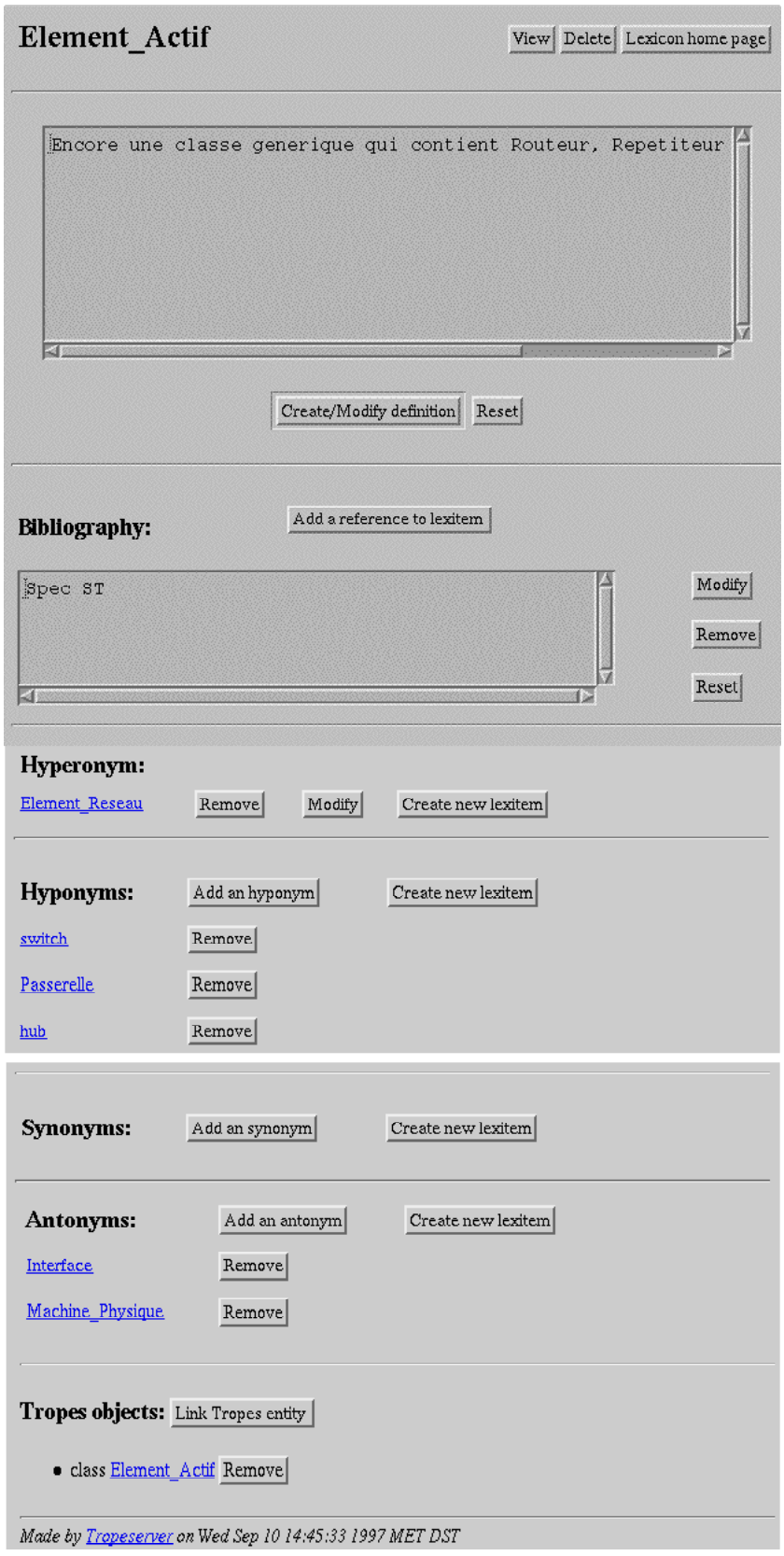


Screendump 37: Lexitem display.

Edit: ..... see Screendump 38, p. 78

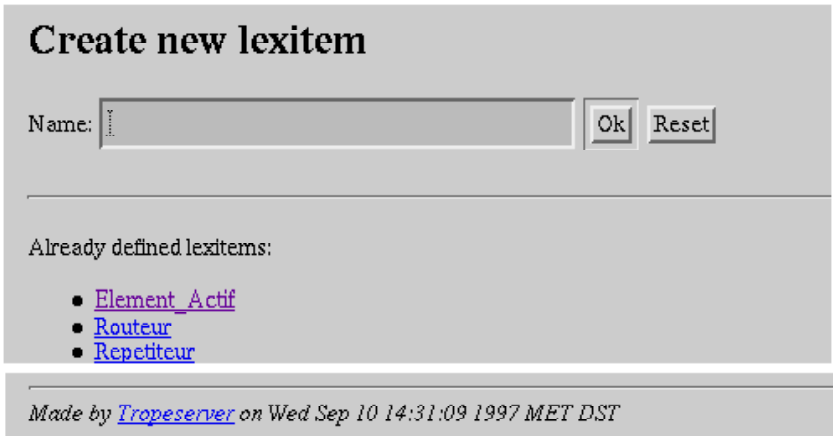
<lexitem>:..... see Screendump 37

<object>:..... see Screendump 31, p.66



Screendump 38: Lexitem edition.

View:..... see Screendump 37, p. 77
<lexitem>:..... see Screendump 38



Screendump 39: Lexitem creation.
OK: ..... see Screendump 38, p. 78

API

(**tr-load-lexicon** *filename optional: print*) function in [libtrp]
-> boolean, *filename* filename$_{naf,fnf}$, *print* boolean
Loads the lexicon contained in file *filename*.

(**tr-save-lexicon** *filename*) function in [libtrp]
-> boolean, *filename* filename$_{naf}$
Saves all the lexitems contained in the environment in the file *filename*.

(**tr-lexitem-p** *anything*) function in [libtrp]
-> boolean, *anything* entity
Returns *anything* if the argument *anything* is a lexitem, `()` otherwise.

(**tr-create-lexitem** *name hyperonym*) function in [libtrp]
-> lexitem, *name* string$_{nas}$, *hyperonyms* ()|lexitem$_{nali}$
Creates a lexitem corresponding to a term. It has *name* for name and *hyperonym* for hyperorym if there are specified.

(**tr-find-lexitem** *name*) function in [libtrp]
-> lexitem, *name* string$_{nas}$
Finds the lexitem corresponding to the term *name*.

(**tr-delete-lexitem** *lexitem*) function in [libtrp]
-> boolean, *lexitem* lexitem$_{nali}$
Deletes the lexitem *lexitem*.

(**tr-lexitem-name** *lexitem*) function in [libtrp]
-> string, *lexitem* lexitem$_{nali}$
Returns the name of the lexitem *lexitem*.

(**tr-lexitem-rename** *lexitem name*) function in [libtrp]
-> string, *lexitem* lexitem$_{nali}$, *name* string$_{nas,adl}$
Returns the name of the lexitem *lexitem*.

(**tr-lexitem-definition** *lexitem*) function in [libtrp]
-> string, *lexitem* lexitem$_{nali}$

Returns the definition of the lexitem *lexitem*.

(**tr-lexitem-hyperonym** *lexitem*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the hyperonym of the lexitem *lexitem*.

(**tr-lexitem-hyponyms** *lexitem*) function in [libtrp]
-> (lexitem*), *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the hyponyms of the lexitem *lexitem*.

(**tr-lexitem-synonyms** *lexitem*) function in [libtrp]
-> (lexitem*), *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the synonyms of the lexitem *lexitem*.

(**tr-lexitem-antonyms** *lexitem*) function in [libtrp]
-> (lexitem*), *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the antonyms of the lexitem *lexitem*.

(**tr-lexitem-bibliography** *lexitem*) function in [libtrp]
-> (string*), *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the bibliographic references of the lexitem *lexitem*.

(**tr-lexitem-objects** *lexitem*) function in [libtrp]
-> (entity*), *lexitem* $\text{lexitem}_{\text{nali}}$
Returns the TROEPS entities related to the lexitem *lexitem*.

(**tr-lexitem-set-hyperonym** *lexitem hyperonym*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *hyper*$\text{lexitem}_{\text{nali}}$
Sets the hyperonym of the lexitem *lexitem* to be the lexitem *hyperonym*.

(**tr-lexitem-set-definition** *lexitem definition*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *definition* $\text{string}_{\text{nas}}$
Sets the definition of the lexitem *lexitem* to be the string *definition*.

(**tr-lexitem-add-hyponym** *lexitem hyponym*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *hyponym* $\text{lexitem}_{\text{nali}}$
Add the lexitem *hyponym* in the hyponyms of the lexitem *lexitem*.

(**tr-lexitem-add-synonym** *lexitem synonym*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *synonym* $\text{lexitem}_{\text{nali}}$
Adds the lexitem *synonym* in the synonyms of the lexitem *lexitem*.

(**tr-lexitem-add-antonym** *lexitem antonym*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *antonym* $\text{lexitem}_{\text{nali}}$
Adds the lexitem *antonym* in the antonyms of the lexitem *lexitem*.

(**tr-lexitem-add-bibliography** *lexitem bibliography*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *bibliography* string
Adds the bibliographic reference *bibliography* in the bibliography of the lexitem *lexitem*.

(**tr-lexitem-add-object** *lexitem object*) function in [libtrp]
-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *object* $\text{entity}_{\text{nao}}$

Adds the entity *object* in the related objects of the lexitem *lexitem*.

(**tr-lexitem-rem-hyponym** *lexitem hyponym*) function in [libtrp]

-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *hyponym* $\text{lexitem}_{\text{nali}}$

Removes the lexitem *hyponym* from the hyponyms of the lexitem *lexitem*.

(**tr-lexitem-rem-synonym** *lexitem synonym*) function in [libtrp]

-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *synonym* $\text{lexitem}_{\text{nali}}$

Removes the lexitem *synonym* from the synonyms of the lexitem *lexitem*.

(**tr-lexitem-rem-antonym** *lexitem antonym*) function in [libtrp]

-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *antonym* $\text{lexitem}_{\text{nali}}$

Removes the lexitem *antonym* from the antonyms of the lexitem *lexitem*.

(**tr-lexitem-rem-bibliography** *lexitem bibliography*) function in [libtrp]

-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *bibliography* string

Removes the bibliographic reference *bibliography* from the bibliography of the lexitem *lexitem*.

(**tr-lexitem-rem-object** *lexitem object*) function in [libtrp]

-> lexitem, *lexitem* $\text{lexitem}_{\text{nali}}$, *object* $\text{entity}_{\text{nao}}$

Removes the entity *object* from the related objects of the lexitem *lexitem*.

(**tr-attach-lexicon** *base*) function in [libtrp]

-> entity, *base* base_{nab}

Attaches to each entity contained in the base base the lexitem corresponding to it.

(**tr-get-lexitem** *entity*) function in [libtrp]

-> lexitem, *entity* $\text{entity}_{\text{nao}}$

Returns the lexitem attached to a particular TROEPS entity *entity* (`()` if none is attached).

Example

```
? (tr-create-lexitem 'house ())
= (tr-lexitem "house")
? (tr-lexitem-name (tr-find-lexitem "house"))
= "house"
? (tr-lexitem-set-definition (tr-find-lexitem "house")
      "a building in which people, usually one familly, live in")
= "a building in which people, usually one familly, live in"
? (tr-lexitem-definition (tr-find-lexitem "house"))
= "a building in which people, usually one familly, live in"
? (tr-lexitem-add-bibliography (tr-find-lexitem "house")
      "Cambridge international dictionary of English, Cambridge university press,
Cambridge (UK), 1995"))
= ("Cambridge international dictionary of English, Cambridge university press, Cambridge
(UK), 1995")
? (tr-lexitem-bibliography (tr-find-lexitem "house"))
= ("Cambridge international dictionary of English, Cambridge university press, Cambridge
(UK), 1995")
? (tr-lexitem-set-hyperonym (tr-find-lexitem "house")
      (tr-create-lexitem 'building ()))
= (tr-lexitem "house")
? (tr-lexitem-hyperonym (tr-find-lexitem "house"))
= (tr-lexitem "building")
? (tr-lexitem-hyponyms (tr-find-lexitem "building"))
= ((tr-lexitem "house"))
? (tr-lexitem-add-hyponym (tr-find-lexitem "building")
      (tr-create-lexitem 'flat ()))
```

```
= ((tr-lexitem "flat")(tr-lexitem "house"))
? (tr-lexitem-hyperonyms (tr-find-lexitem "flat"))
= ()
? (tr-lexitem-hyponyms (tr-find-lexitem "building"))
= ((tr-lexitem "flat")(tr-lexitem "house"))
? (tr-lexitem-add-synonym (tr-find-lexitem "flat")
      (tr-create-lexitem 'appartment ()))
= ((tr-lexitem "flat"))
? (tr-lexitem-synonyms (tr-find-lexitem "flat"))
= ((tr-lexitem "appartment"))
? (tr-lexitem-synonyms (tr-find-lexitem "appartment"))
= ((tr-lexitem "flat"))
? (tr-lexitem-add-antonym (tr-create-lexitem 'furnished ()))
      (tr-create-lexitem 'non furnished ()))
= ((tr-lexitem "furnished"))
? (tr-lexitem-antonyms (tr-find-lexitem "furnished"))
= ((tr-lexitem "non furnished"))
? (tr-lexitem-antonyms (tr-find-lexitem "non furnished"))
= ((tr-lexitem "furnished"))
? (tr-attach-lexicon (tr-base "real-estate"))
= t
? (tr-lexitem-objects (tr-find-lexitem "house"))
= ((tr-class (tr-conceptview (tr-concept "house") "price") "house")
   (tr-class (tr-conceptview (tr-concept "house") "test") "house")
   (tr-class (tr-conceptview (tr-concept "house") "size") "house")
   (tr-class (tr-conceptview (tr-concept "house") "localisation") "house")
   (tr-concept "house"))
? (tr-get-lexitem (tr-concept "house"))
= (tr-lexitem "house")
? (tr-lexitem-add-object (tr-find-lexitem "house")
      (tr-object (tr-concept "house") '("rue Monge" 2 3))))
= (tr-lexitem "house")
? (tr-lexitem-objects (tr-find-lexitem "house"))
= ((tr-object (tr-concept "house") '("rue Monge" 2 3))
   (tr-class (tr-conceptview (tr-concept "house") "price") "house")
   (tr-class (tr-conceptview (tr-concept "house") "test") "house")
   (tr-class (tr-conceptview (tr-concept "house") "size") "house")
   (tr-class (tr-conceptview (tr-concept "house") "localisation") "house")
   (tr-concept "house"))
? (tr-save-lexicon #f"real-estate.ldf")
= ()
? (tr-lexitem-rem-object (tr-find-lexitem "house") (tr-concept "house"))
= (tr-lexitem "house")
? (tr-lexitem-objects (tr-find-lexitem "house"))
= ((tr-object (tr-concept "house") '("rue Monge" 2 3))
   (tr-class (tr-conceptview (tr-concept "house") "price") "house")
   (tr-class (tr-conceptview (tr-concept "house") "test") "house")
   (tr-class (tr-conceptview (tr-concept "house") "size") "house")
   (tr-class (tr-conceptview (tr-concept "house") "localisation") "house"))
? (tr-lexitem-rem-synonym (tr-find-lexitem "flat") (tr-find-lexitem "appartment"))
= ()
? (tr-lexitem-synonyms (tr-find-lexitem "appartment"))
= ()
? (tr-lexitem-rem-antonym (tr-find-lexitem "furnished") (tr-find-lexitem "non furnished"))
= ()
? (tr-lexitem-antonyms (tr-find-lexitem "non furnished"))
= ()
? (tr-lexitem-rem-bibliography (tr-find-lexitem "house")
      "Cambridge international dictionary of English, Cambridge university press,
Cambridge (UK), 1995")
= ()
? (tr-lexitem-bibliography (tr-find-lexitem "house"))
= ()
? (tr-lexitem-rem-hyponym (tr-find-lexitem "building") (tr-find-lexitem "flat"))
= ()
? (tr-lexitem-hyperonym (tr-find-lexitem "flat"))
= ()
? (tr-delete-lexitem "house")
= t
```

```
? (tr-find-lexitem "house")
= ()
? (tr-load-lexicon #f"real-estate.ldf")
= ()
? (tr-find-lexitem "house")
= (tr-lexicon "house")
```

Computing field values

Several mechanisms will be provided in TROEPS for computing field values: constraints, default inferences and tasks.

Implicit versus hypothetical computation

Not all the information about the objects is explicitly provided. In fact, some slot can remain unfilled (or undefined). TROEPS provides mechanisms for computing the values of these fields. However, they are not all at the same level because leaving a field unfilled can have several meanings:

- It can mean that the field has no value for the object;
- It can mean that the field value is unknown to the users;
- It can mean that the field value is known to the users to be the value that TROEPS will compute and thus the users do not care about filling it.

These interpretation are different but usually, knowledge representation systems do not distinguish them. In TROEPS, there are computation mechanisms for the second and the third item and they are clearly distinguished. Concerning the second meaning, since the users do not know the value, the system will only be able to compute hypothetical values, i.e. guesses based on the particular position of the object in the taxonomies. Concerning the third meaning, TROEPS is able to compute an implicit value which is the true value of the object (provided that the user has described the methods for computing the value). For instance, imagine that, in the knowledge base presented above, it is considered that if no value is given to the field `second-hand`, this means that the corresponding house is second hand (and thus the field value is `true`). This can be used at input time in order to fill only the houses which are brand new ones (implicit value) or this can be used when the status of the house is unknown that it is a second hand house (hypothetical value, which seems statistically reasonable since there are more second hand houses to sell). These are two different interpretation which, in turn, induce totally different behaviour from the users of the knowledge base.

There is a third kind of computing mechanisms called constraints which are not available in the present version of TROEPS.

These are not the only differences between hypothetical and implicit computations. As usual, the implicit computations are part of the “ontological aspect” of the objects and thus are declared in the concept. As a matter of fact, since these implicit computations return the true value of the field, this value does not depend on the classes the object is attached to. The user can declare several methods in order to compute the value but they are supposed to all return the same value and thus only one value is returned by the system. Lastly, since the value is the true value, it is available by using the `tr-get-value` primitive.

Hypothetical values are guesses depending on the position of the object in the taxonomies. Thus, the inference mechanisms are declared in the classes. For the same object, they can return different values which are concurrent guesses. The hypothetical values are computed on demand through the `tr-guess-value` primitive. Since, there can be several hypothetical values (at most one per viewpoint), `tr-guess-value` returns a list of values (ordered in the same way the viewpoints are ordered in the call). If no hypothetical value has been found, it returns the undefined value.

Each mechanism available in TROEPS is described below. There exists only one kind of implicit computation called "procedural detachment" and which, in the current version of TROEPS is mere function call. There exist four kinds of hypothesising mechanisms: default values, value passing, procedural detachment (again, declared in classes) and filtering. These mechanisms are parallel to four exact value computation mechanisms available in TROEPS or other systems:

Mechanism	TROEPS exact computation	SHIRKA equivalent
Default value	Type reduced to a singleton (`$value` in SHIRKA)	`$default`
Value passing	Constraint	`$var<-`
Procedural detachment	Task invocation	`$sib-exec`
Filters	Filtering	`$sib-filtre`

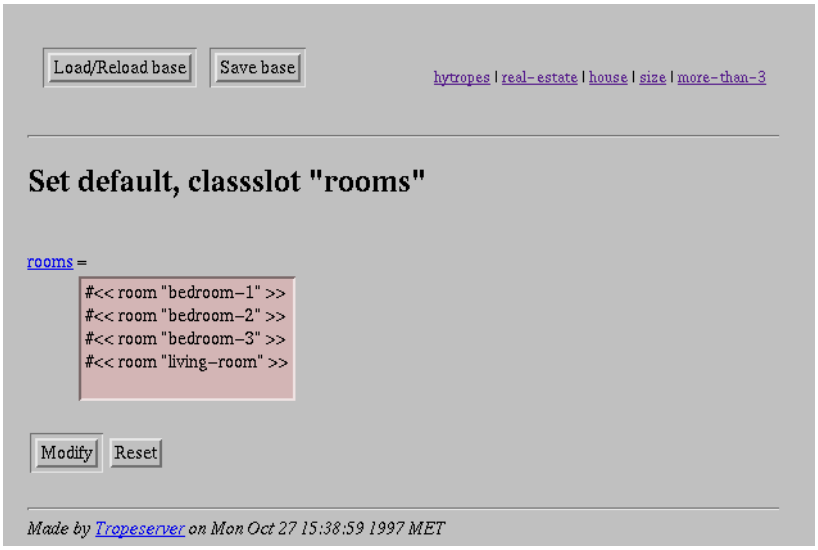
Hypothetical computation, inheritance by default and ordering

The mechanisms for guessing hypothetical values are inherited by default in the taxonomy. This means that TROEPS will try to obtain the guesses from all the classes the object is attached to. If no mechanism attached to one of these classes returns a value, it will try again with their super-classes and so on so forth. However, as soon as a value is returned, the process will stop with that value (for the taxonomy). If other mechanisms are defined in super-classes they are shadowed by those defined in more specific classes (it is supposed that if they are more specific they are able to provide more adapted guesses).

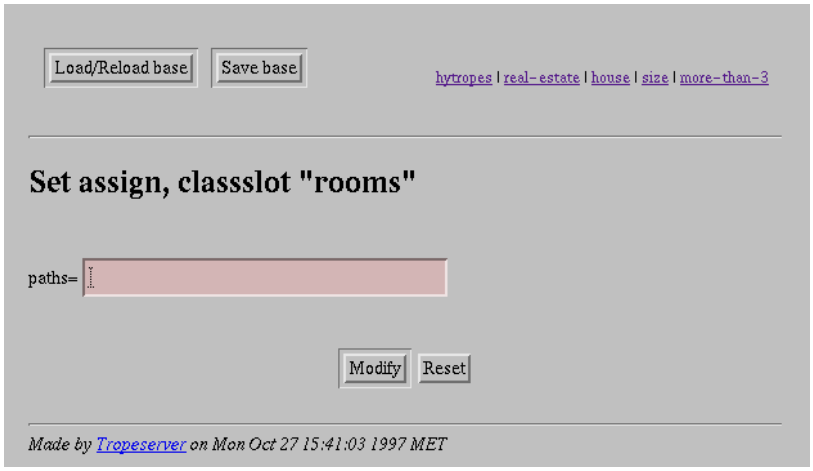
The computation of hypothetical values is not easy to follow and is thus provided here. However, it is not necessary to understand it for using hypothetical computations. The algorithm is roughly the following:

1. When `tr-guess-value` is called with several viewpoints as arguments it first considers independently each viewpoint in the same order.
2. In each taxonomy it tries to find a value by investigating the successive super-classes of the one the object is attached to. If a value is obtained in a class, this value is returned and no other class is considered.
3. In each class it tries each inference descriptor in the following order: `assign`, `filters`, `methods` and `default`. If a value is obtained by a descriptor, this value is returned and no other descriptor is considered.
4. In each descriptor there can be several method (but for default) and the system tries each one in the order in which they are stored in the descriptor. The obtained values are checked if they satisfy the particular constraints applying to the object. If a value is obtained by a method and satisfy the constraints, this value is returned and no other method is considered.
5. For each method, the constraints of filters and the arguments of procedural detachment must be computed before triggering them. This can require additional hypothetical computation which is triggered exclusively in the current viewpoint. If this computation fails to return a value, then the method fails.

As a result, `tr-guess-value` returns at most one value by viewpoint.



Screendump 40: Class field default value modification.
Modify: ..... see Screendump 19, p. 48
Reset: ..... see Screendump 40



Screendump 41: Class field assign modification.
Modify: ..... see Screendump 19, p. 48
Reset: ..... see Screendump 41

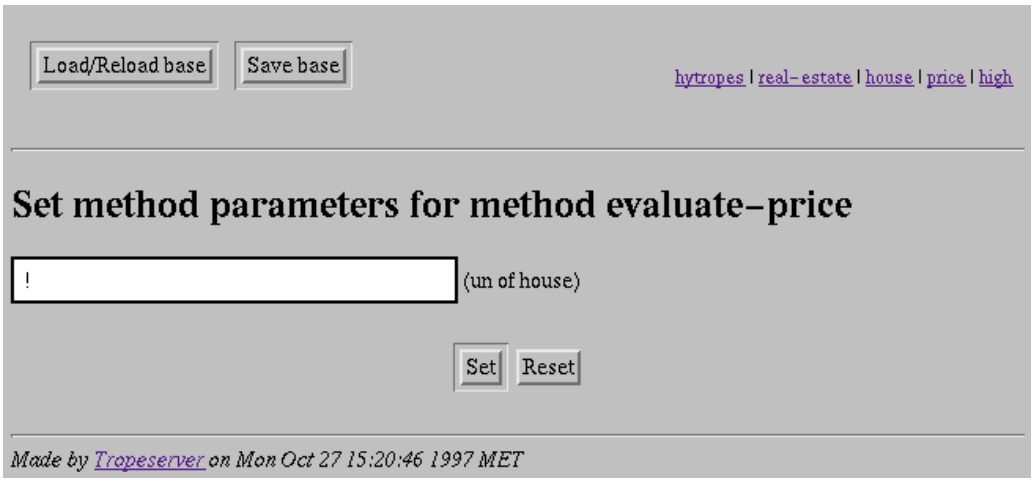
NIY

Screendump 42: Class field filter modification.
Modify: ..... see Screendump 19, p. 48
Reset: ..... see Screendump 42

Screendump 43: Class field method modification.

Add:..... see Screendump 44, p. 87

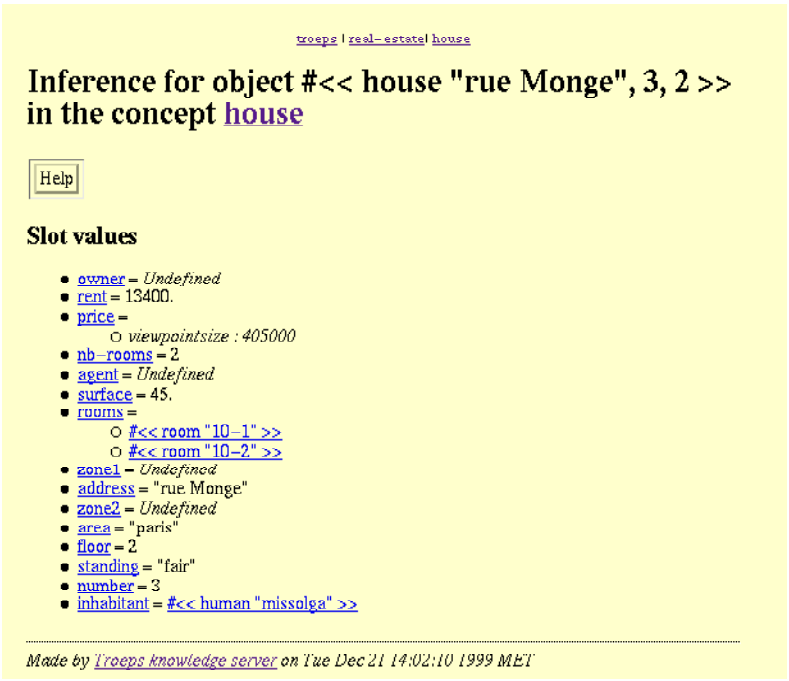
Reset: ..... see Screendump 44



Screendump 44: Class field method argument modification.

Set: ..... see Screendump 19, p. 48

Reset: ..... see Screendump 44



Screendump 45: Guessed value display.
<slot>:.....see Screendump 8, p. 29
<object>:.....see Screendump 31, p. 66

API

(**tr-guess-value** *object slotname rest: views*) function in [libtrp]

-> {(conceptview . value)*}, *object* $\text{object}_{\text{nao}}$, *slotname* $\text{string}_{\text{nsa, ukks}}$, *views* { $\text{conceptview}_{\text{nakv}}$* }$_{\text{ivsp}}$

If the field of name *slotname* of object *object* has a value, this value is returned. Otherwise, uses the inference methods stored in the method descriptors of the different classes (under the considered viewpoints) for returning the hypothetical values under each viewpoint.

Default value

The more basic inference means consists in using a default value for a particular field. The default value is provided in classes and remains valid for the sub-classes as far as no other default value is provided. The example above give the default value `second-hand` as `true` for all the houses. However, if a new area is known to have been just recently constructed, then the objects attached to the class corresponding to that area will have the `false` default value for the field. Then, if an objects is attached to that class the guessed value will be `false`.

The default classslot descriptor allows to provide a default value for a field which must be an acceptable value for the class.

API

(**tr-get-classslot-default** *classslot*) function in [libtrp]

-> value, *classslot* $\text{classslot}_{\text{nacs}}$

Returns the default value defined and applicable to the class field *classslot*.

(**tr-set-classslot-default** *classslot value*) function in [libtrp]

-> boolean, *classslot* $\text{classslot}_{\text{nacs}}$, *value* $\text{entity}_{\text{ivsv, kcv}}$

Sets the default value applicable to the class field *classslot*.

(**tr-unset-classslot-default** *classslot*) function in [libtrp]

-> boolean, *classslot* $\text{classslot}_{\text{nacs}}$

Removes the default value applicable to the class field *classslot*.

Examples

```
? (tr-get-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "standing")
= #s(<undef>)
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "standing"
      (tr-concepview (tr-concept "house") "price"))
= #s(<undef>)
? (tr-set-classslot-default
      (tr-classslot (tr-conceptview-root (tr-concepview (tr-concept "house") "price"))
"standing")
      "high")
= (((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
? (tr-get-classslot-default
      (tr-classslot (tr-conceptview-root (tr-concepview (tr-concept "house") "price"))
"standing"))
= "high"
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "standing"
      (tr-concepview (tr-concept "house") "price"))
= (((tr-conceptview (tr-concept "house") "price") . "high"))
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "standing"
```

```
      (tr-concepview (tr-concept "house") "localisation"))
= (((tr-conceptview (tr-concept "house") "localisation") . #s(<undef>)))
? (tr-set-classslot-default
      (tr-classslot
         (tr-is-in-under (tr-object (tr-concept "house") `("rue Monge" 2 3))
                         (tr-concepview (tr-concept "house") "localisation"))
         "standing")
      "low")
= "low"
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "standing")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation"). "low")
   ((tr-conceptview (tr-concept "house") "size") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . "high"))
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "standing"
      (tr-concepview (tr-concept "house") "price"))
= (((tr-conceptview (tr-concept "house") "price") . "high"))
? (tr-unset-classslot-default
      (tr-classslot     (tr-conceptview-root     (tr-conceptview     (tr-concept     "house")
"localisation")) "standing"))
= #s(<undef>)
? (tr-set-classslot-default
      (tr-classslot     (tr-conceptview-root     (tr-concepview     (tr-concept     "house")
"localisation")) "standing")
      "high")
= "high"
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "standing")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation") . "high")
   ((tr-conceptview (tr-concept "house") "size") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . "high"))
```

Paths and slot value passing

Another inference mechanism consists in looking for a plausible value in another field of the same object, or a field of another object which is referred to by the former one. As an example, it could be plausible that someone has the same name as his wife. So the name field can be looked for in the name field of his wife.

This is achieved by specifying a path from the object to where the value can be found. This path is the succession of fields to be used in order to access the value. In the example above, the path is expressed by `!.wife` in which "`!`" stands for the object itself and wife is the name of a field. This can be iterated such that `!.wife.sons` are the sons of the wife and `!.wife.sons.wife` are the wives of the sons of the wife (daughters in law).

The definition of paths can lead to strange values. For instance, if one tries to define the set of grand-sons by the path `!.children.sons`, the result should be a set of set of persons (one set per child). It should be better to return just a set of persons (because this is a valid slot value). This is possible through the ":" construct `!::children.sons` returns the union of the children of the sons of the person.

Now, the assign classslot descriptor allows to provide a value passing specification for a field. The values in the assign descriptor are valid paths from the current class (i.e. paths which can be followed from the instances of the class and which lead to a value of the correct type).

When looking for an hypothetical value for a field, if the assign descriptor contain paths, the paths are followed (in the order of the descriptor content) and the first one to return a value will be considered as the guess. It is noteworthy that there are several paths because a path can fail to return a value if one of the elements along the path is missing.

API

(**tr-make-path** *conceptslots*) function in [libtrp]

-> path, *conceptslots* {conceptslot*}$_{nacs}$

Creates and returns a path starting at the concept of the first element of conceptslots and following the sequence of these conceptslots.

(**tr-path-p** *anything*) function in [libtrp]

-> boolean, *anything* entity

Returns *anything* if the argument *anything* is a path, `()` otherwise.

(**tr-get-classslot-assigns** *classslot*) function in [libtrp]

-> {path*}, *classslot* classslot$_{nacs}$

Returns the set of path assignments defined and applicable to the class field *classslot*.

(**tr-add-classslot-assign** *classslot paths*) function in [libtrp]

-> boolean, *classslot* classslot$_{nacs}$, *paths* (path*)

Adds a path to those defined and applicable to the class field *classslot*.

(**tr-rem-classslot-assign** *classslot paths*) function in [libtrp]

-> boolean, *classslot* classslot$_{nacs}$, *paths* (path*)

Removes a path from those applicable to the class field *classslot*.

Examples

```
? (tr-get-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "inhabitant")
= #s(<undef>)
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "inhabitant"
      (tr-concepview (tr-concept "house") "size"))
= (((tr-conceptview (tr-concept "house") "size") . #s(<undef>)))
? (tr-add-classslot-assign (tr-classslot (tr-conceptview-root (tr-conceptview (tr-concept
"house") "size")) "inhabitant") (tr-make-path (tr-concept "house") "!.owner"))
= ((tr-make-path (tr-concept "house") "!.owner"))
? (tr-get-classslot-assign (tr-classslot (tr-conceptview-root (tr-conceptview (tr-concept
"house") "size")) "inhabitant")
= ((tr-make-path (tr-concept "house") "!.owner"))
?  (tr-path-p  (car  (tr-get-classslot-assign  (tr-classslot  (tr-conceptview-root  (tr-
conceptview (tr-concept "house") "size")) "inhabitant"))
= ((tr-make-path (tr-concept "house") "!.owner"))
? (tr-set-value (tr-object (tr-concept "house") '("rue Monge" 2 3))
   "owner"
   (tr-create-object (tr-concept "human") '("M.M")))
= (tr-object (tr-concept "human") '("M.M"))
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "inhabitant")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation")
      (tr-object (tr-concept "human") '("M.M")))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
? (tr-rem-classslot-assign (tr-classslot (tr-conceptview-root (tr-conceptview (tr-concept
"house")  "size"))  "inhabitant")  (car  (tr-get-classslot-assign  (tr-classslot  (tr-
conceptview-root (tr-conceptview (tr-concept "house") "size")) "inhabitant"))))
= ()
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "inhabitant"))
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation"). #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
```

Computation by filter

The computation by filters can be used by set valued slots. It consists in creating a filter which returns the set of filtered values which will then be returned as the guessed value for the field. These filters can support additional constraints provided through the usual filter and class

manipulation primitives. Moreover, these constraints can be computed by inference descriptors which are defined with regard to a current object.

The `filters` classslot descriptor allows to provide filters for computing a field.

When a value is required and a filter computation is necessary, first the actual constraints of the filter are computed with regard to the current object and then the filtered object are obtained as usual (see the "Filters" section in the previous part of the manual).

API

(**tr-get-classslot-filters** *classslot*) function in [libtrp]

-> { filter* }, *classslot* classslot$_{nacs}$

Returns the set of filters defined for and applicable to the class field *classslot*.

(**tr-add-classslot-filters** *classslot filters*) function in [libtrp]

-> boolean, *classslot* classslot$_{nacs}$, *filters* (filter*), adf

Add a new filter to the set of filters defined for and applicable to the class field *classslot*.

(**tr-rem-classslot-filters** *classslot filters*) function in [libtrp]

-> boolean, *classslot* classslot$_{nacs}$, *filters* (filter*), adf

Removes a filter from the set of filters defined for and applicable to the class field *classslot*.

Example

```
? (tr-get-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "rooms")
= #s(<undef>)
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "rooms"
      (tr-concepview (tr-concept "house") "size"))
= (((tr-conceptview (tr-concept "house") "size") . #s(<undef>)))
? (tr-add-classslot-filter
   (tr-classslot
      (tr-conceptview-root (tr-conceptview (tr-concept "house") "size")) "rooms")
   (tr-create-filter (tr-concept "room")))
= (tr-filter (tr-concept "room"))
? (tr-get-classslot-filters
   (tr-classslot
      (tr-conceptview-root (tr-conceptview (tr-concept "house") "size")) "rooms"))
= ((tr-filter (tr-concept "room")))
? (tr-create-complete-object (tr-concept "room") ()
      (list (cons "id" "rm12")
            (cons "appart" (tr-object (tr-concept "house") `("rue Monge" 2 3)))))
= (tr-object (tr-concept "room") `("rm12"))
? (tr-create-complete-object (tr-concept "room") ()
      (list (cons "id" "rm13")
            (cons "appart" (tr-object (tr-concept "house") `("rue Monge" 2 3)))))
= (tr-object (tr-concept "room") `("rm13"))
? (tr-create-complete-object (tr-concept "room") ()
      (list (cons "id" "rm14")
            (cons "appart" (tr-object (tr-concept "house") `("rue Monge" 2 3)))))
= (tr-object (tr-concept "room") `("rm14"))
? (tr-rem-classslot-filter
   (tr-classslot
      (tr-conceptview-root (tr-conceptview (tr-concept "house") "size")) "rooms")
   (car (tr-get-classslot-filter
         (tr-classslot
            (tr-conceptview-root (tr-conceptview (tr-concept "house") "size")) "rooms"))))
= ()
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "rooms")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation"). #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
```

Task invocation

Task or method invocation covers what has been called procedural detachment and the use of a function in order to compute a value. The tasks are actually simple (C or talk) functions invoked on their arguments and returning a value. They are described through the generic term « method ».

The `methods` classslot or conceptslot descriptor allows to provide a task specification for a field. These specification follows the same line, they consists in a list beginning by the method name (the name of the function to be called) and the list of its argument specification (in the correct order). These arguments can be either concrete values or paths.

The tasks are computed in the order under which they are in the descriptor. The computation stops with the first succeeding call. Before computing the task, the paths taken as arguments are evaluated. If any of them return no value then the task invocation fails. Once the arguments are computed, the function is called against them. It returns either undefined (`(unbound-slot-value)`) or another value. In the first event, the task fails, in the second one, the value is returned.

Before they can be used, the methods must be declared with their signature. This allows static type checking of the method when it is put in the field. The concepts involved in the method type do not have to be declared when the method is declared (so this can be done before loading the base for instance) but they must have been loaded when it is added to a field.

API

(**tr-declare-method** *name args result*) function in [libtrp]

-> symbol, *name* $\text{symbol}_{\text{aem}}$, *args* $((\text{string . string})^*)_{\text{naks}}$, *result* $(\text{string . string})_{\text{naks}}$

Declares a mehod of name *name* and signature *args -> result*. The signature is made of couple constructor/type as for defining a conceptslot.

(**tr-get-conceptslot-methods** *conceptslot*) function in [libtrp]

-> $((\text{symbol } (\text{entity}|\text{path})^*)^*)$, *conceptslot* $\text{conceptslot}_{\text{naks}}$

Returns the set of methods attached to the concept field *conceptslot*.

(**tr-add-conceptslot-method** *conceptslot method*) function in [libtrp]

-> boolean, *conceptslot* $\text{conceptslot}_{\text{naks}}$, *method* $(\text{symbol}_{\text{nas}} (\text{entity}|\text{path})^*)_{\text{ivm}}$

Adds the new inference methods *methods* for the concept field *conceptslot*. The method specification *method* is a list begining by the method name followed by the method arguments. The arguments are either values or paths. They return either a value for the corresponding field of the object, either the result of the function `(unbound-slot-value)`. When necessary, these methods are evaluated from the first to the last in this list.

(**tr-rem-conceptslot-method** *conceptslot method*) function in [libtrp]

-> boolean, *conceptslot* $\text{conceptslot}_{\text{naks}}$, *method* $(\text{symbol}_{\text{nas}} (\text{entity}|\text{path})^*)_{\text{ivm}}$

Removes the method *method* from those attached to the concept field *conceptslot*.

(**tr-get-classslot-methods** *classslot*) function in [libtrp]

-> $((\text{symbol } (\text{entity}|\text{path})^*)^*)$,*classslot* $\text{classslot}_{\text{nacs}}$

Returns the set of methods defined for and applicable to the class field *classslot*.

(**tr-add-classslot-method** *classslot method*) function in [libtrp]

-> boolean, *classslot* $\text{classslot}_{\text{nacs}}$, *methods* $((\text{symbol}_{\text{nas}} (\text{entity}|\text{path})^*)_{\text{ivm}})^*$

Sets the new inference methods *methods* for the class field *classslot*. The method specification *method* is a list begining by the method name followed by the method arguments. The arguments are either values or paths. They return either a value for the corresponding field of the object, either the result of the function `(unbound-slot-value)`. When necessary, these methods are evaluated from the first to the last in this list.

(**tr-rem-classslot-method** *classslot method*) function in [libtrp]

-> boolean, *classslot* $\text{classslot}_{\text{nacs}}$, , *methods* $((\text{symbol}_{\text{nas}}\ (\text{entity}|\text{path})^*\ )_{\text{ivm}})^*$

Removes the inference methods *methods* for the class field *classslot*.

Example

```
? (tr-get-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "price")
= #s(<undef>)
? (tr-guess-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "price"
      (tr-concepview (tr-concept "house") "size"))
= (((tr-conceptview (tr-concept "house") "size") . #s(<undef>)))
? (defun evaluate-price (house)
      (if (undefinedp (tr-get-value house "surface"))
         (unbound-slot-value)
         (let ((basis (* (tr-get-value house "surface") 9000)))
            (cond
               ((tr-superclass-p
                  (tr-class (tr-conceptview (tr-concept "house") "size") "more-than-3")
                  (tr-is-in-under house (tr-conceptview (tr-concept "house") "size")))
                  (int (* basis 0.9)))
               ((tr-superclass-p
                  (tr-class (tr-conceptview (tr-concept "house") "size") "more-than-3")
                  (tr-is-in-under house (tr-conceptview (tr-concept "house") "size")))
                  (int (* basis 1.2)))
               (t (int basis))))))
= evaluate-price
? (tr-declare-method 'evaluate-price
        (list (cons "un" "house"))
        (cons "un" "entier"))
= evaluate-price
? (tr-add-classslot-method
   (tr-classslot (tr-conceptview-root
                     (tr-conceptview (tr-concept "house") "size"))
                 "price")
   (list 'evaluate-price (tr-make-path (tr-concept "house") "!")))
= ((evaluate-price (tr-make-path (tr-concept "house") "!")))
? (tr-get-classslot-methods
      (tr-classslot
         (tr-conceptview-root (tr-conceptview (tr-concept "house") "size"))
         "price"))
= ((evaluate-price (tr-make-path (tr-concept "house") "!")))
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "price")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "size") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
? (tr-set-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "surface" 45.)
= 45.
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "price")
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "size") . 405000)
   ((tr-conceptview (tr-concept "house") "localisation") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
? (tr-rem-classslot-method
      (tr-classslot
         (tr-conceptview-root (tr-conceptview (tr-concept "house") "size"))
         "price")
      (car (tr-get-classslot-methods
            (tr-classslot
```

```
              (tr-conceptview-root (tr-conceptview (tr-concept "house") "size"))
            "price"))))
= ()
? (tr-guess-value (tr-object (tr-concept "house") '("rue Monge" 2 3)) "price"))
= (((tr-conceptview (tr-concept "house") "test") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "localisation") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "size") . #s(<undef>))
   ((tr-conceptview (tr-concept "house") "price") . #s(<undef>)))
? (tr-get-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "nb-rooms")
= #s(<undef>)
? (defun re-llen (list)
    (if (undefinedp list) (unbound-slot-value) (llen list)))
= re-llen
? (tr-declare-method 're-llen
        (list (cons "ensemble" "room"))
        (cons "un" "entier"))
= re-llen
? (tr-add-conceptslot-method
    (tr-find-conceptslot (tr-find-concept "house") "nb-rooms")
      (list 're-llen (tr-make-path (tr-concept "house") "!.rooms")))
= ((re-llen (tr-make-path (tr-concept "house") "!.rooms")))
? (tr-get-conceptslot-methods
    (tr-find-conceptslot (tr-find-concept "house") "nb-rooms"))
= ((re-llen (tr-make-path (tr-concept "house") "!.rooms")))
? (tr-get-value (tr-object (tr-concept "house") `("rue Monge" 2 3)) "nb-rooms")
= 2
```

Constraints

While an object-based knowledge representation system like TROEPS is well-suited to describe, organise, infer and store large amounts of knowledge, there is also a need both for reasoning using field domains (and not only field values, like methods do) and for expressing relations (and not only the "part-of" relation) or properties about objects.

In order to achieve these two goals in TROEPS, the constraint paradigm seems to provide a helpful and powerful tool. Indeed, a constraint $C(x_1,\ldots x_n)$ describes a relation that must be satisfied by a set $V = \{x_1,\ldots x_n\}$ of variables (named the constrained variables), each x_i having an associated domain D_i of values. In usual constraint programming languages, given a set of constraints, a constraint solver applies satisfaction algorithms (to eliminate from each domain, the values that can not appear in any solution) in order to find one, some or all the solutions of this constraint satisfaction problem.

Constraints are not available within this version of TROEPS.

Constraint constructions

The constraints (or better constraint expressions) are build from constraints predicates (expressing a constraints between several values), constraint operators (expressing the construction of a value from others), paths (designating the variables) and values. For instance:

`=`	is a constraint predicate (a = b is either true or false),
`*`	is a constraint operator (constructing an integer from other integers),
`12`	is a value, and
`!.monthly-rent`	is a path

The expression `(* !.monthly-rent 12)` is called a constraint term whose operator is `*` and `(= !.yearly-rent (* !.monthly-rent 12))` is called a constraint. It constrains two expressions to be equal.

Constraints as unary predicates

Typing descriptors reveal themselves inadequate for stating a property the value of an field has to satisfy. Such a property can now be expressed with a unary predicate given by the new descriptor `condition`.

Constraint library

API

(**tr-make-constraint** *constraint-predicate* rest: *constrained-terms*)— function in [libtrp]
-> constraint, *constraint-predicate* contraint-predicate$_{nacp}$, *constrained-terms* (constrained-term|path|entity *)$_{nace}$ NIY

Creates and returns a constraint expression representing the application of a constraint predicate against its list of arguments.

(**tr-make-constraint-term** *constraint-operator* rest: *constrained-terms*)— function in [libtrp]
-> constraint-term, *constraint-operator* contraint-operator$_{naco}$, *constrained-terms* (constrained-term|path|entity*)$_{nace}$ NIY

Creates and returns a constraint term representing the application of a constraint operator against its list of arguments.

(**tr-concept-add-constraint** *concept constraint*)— function in [libtrp]
-> boolean, *concept* concept$_{nak}$, *constraint* constraint$_{nact}$ NIY

Adds the constraint *constraint* to the set of constraints applying to the concept *concept*.

(**tr-concept-rem-constraint** *concept constraint-exp*)— function in [libtrp]
-> boolean, *concept* concept$_{nak}$, *constraint* constraint$_{nact}$ NIY

Suppresses the constraint *constraint* from the set of constraints applying to the concept *concept*.

(**tr-concept-constraints** *concept*)— function in [libtrp]
-> { constraint* }, *concept* concept$_{nak}$ NIY

Returns the constraints applying to the concept *concept*.

(**tr-class-add-constraint** *class constraint*)— function in [libtrp]
-> boolean, *class* class$_{nac}$, *constraint* constraint$_{nact}$ NIY

Adds the constraint *constraint* to the set of constraints applying to the concept *concept*.

(**tr-class-rem-constraint** *class constraint*)— function in [libtrp]
-> boolean, *class* class$_{nac}$, *constraint* constraint$_{nact}$ NIY

Suppresses the constraint *constraint* from the set of constraints applying to the concept *concept*.

(**tr-class-constraints** *class*)— function in [libtrp]
-> { constraint* }, *class* class$_{nac}$ NIY

Returns the constraints applying to the class *class*.

(**tr-object-add-constraint** *object constraint*)— function in [libtrp]
-> boolean, *object* object$_{nao}$, *constraint* constraint$_{nact}$ NIY

Adds the constraint expressed by *constraint* to the set of constraints applying to the object *object.*

(**tr-object-rem-constraint** *object constraint*)— function in [libtrp]
-> boolean, *object* object$_{nao}$, *constraint* constraint$_{nact}$ NIY

Deletes the constraint *constraint* to the set of constraints applying to the object *object*.

(**tr-object-constraints** *object*)— function in [libtrp]
-> { constraint* }, *object* object$_{nao}$ NIY

Returns the constraints applying to the class *class*.

(**tr-get-classslot-condition** *classslot*)— function in [libtrp]
-> { constraint-predicate* }, *classslot* classslot$_{nacs}$ NIY

Returns the conditions of the class field *classslot*, `()` if they are not refined in the class field.

(**tr-set-classslot-condition** *classslot condition*)— function in [libtrp]
-> boolean, *classslot* classslot$_{nacs}$, *condition* constraint-predicate$_{nacp}$ NIY

Refines the conditions of the class field *classslot* by restricting them to the new conditions *condition*.

Example

Classification

The taxonomies used by TROEPS can be manipulated in order to find adequate places for objects and classes in a given category or for building (upon particular criteria) a taxonomy from the instances of the concept. Each functionality is presented in detail before giving a complete example.

Instance classification

Classification is a suggestion mechanism, but instead of guessing the value of an object field, it tries to find the possible classes for an object to migrate from its current class to a lower (i.e. more specific) class. It is based on matching the field values of the instance to be classified with the current candidate class description. The classification takes place in each viewpoint along the suitable hierarchy and uses bridges to jump from one class in a viewpoint to another one in another viewpoint.

Starting from the current belonging class of the instance in a given viewpoint, the classification mechanism scans all of its subclasses and determines, for each of them, if it is possible, unknown or impossible. A class is:

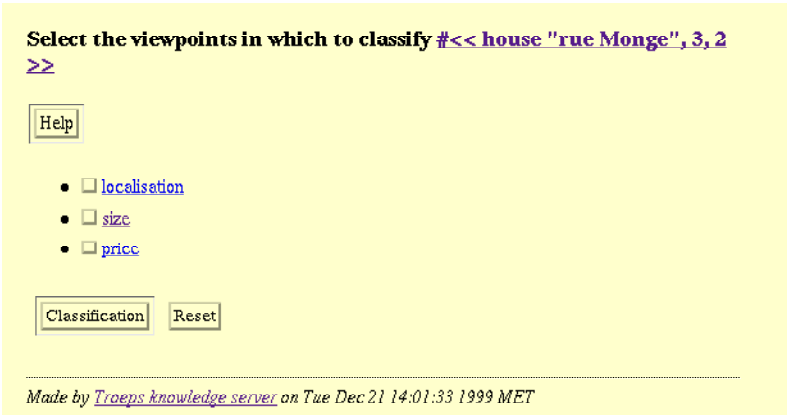
- possible if the instance satisfies all the constraints on all the fields that exist in this class;
- it is unknown if no constraint in the class is violated, i.e. when the instance is incomplete (with regard to the class specifications);
- it is impossible if at least one constraint is not satisfied by a field value.

When a class reveals itself impossible, all its sub-classes are labelled impossible too. Thus, the algorithm iterates on every possible or unknown sub-classes, and ends after it has reached the leaves of the hierarchy. When the classification labels a class which is also the only source of a bridge, as possible or unknown, it deduces that the destination class of the bridge (in another viewpoint) is possible or unknown too. If this is not possible (because the object does not satisfy the constraints of the destination), the initial object is finally labelled as impossible.

The function provided below constitutes the minimal classification service in TROEPS. Improvements are planned including:

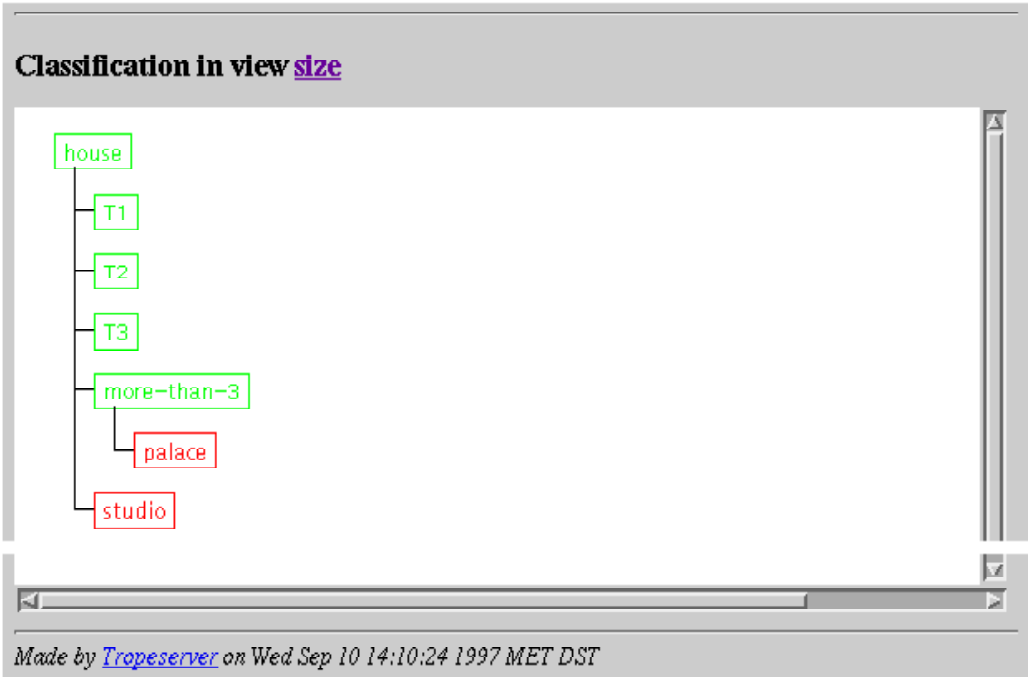
- the interactive querying of the user for unknown values (with, if possible, clever algorithms);
- taking into account exclusive, exhaustive and definitional nature of the taxonomies;
- better algorithm for classification cache;
- taking into account recursive objects.

Interface



Screendump 46: Classification initialisation.

<view>: ..... see Screendump 11, p. 34
Classify:..... see Screendump 47, p. 98
Reset: ..... see Screendump 46



Screendump 47: Classification result. Classes are displayed in either red (impossible), yellow (unknown) or green (possible). As a result of classification, clicking on a class will access to that class while when used in attach query (see Screendump 32), it will attach the object to the selected class.

<class>: ..... see Screendump 14, p. 40

API

(**tr-classify** *object rest: views*) function in [libtrp]

-> (((class*) (class*) . (class*))*), *object* object_{nao},*views* { $\text{conceptview}_{nakv}$* }$_{ivsp}$

Returns, for each of the provided concept viewpoints *views* — and if none is provided, for each of the views of the concept —, the sets of possible, unknown and impossible classes for the object *object*.

(**tr-attachable** *object class*) function in [libtrp]

-> symbol: {imp, unk, pos}, *object* object_{nao},*class* $\text{class}_{nac,ivc}$

Returns a value among `pos`, `unk` and `imp` indicating if the object *object* is attachable to the class *class*. Note that if the class *class* is not a subclass of the current object class under the concept, the returned value is `imp`.

Examples

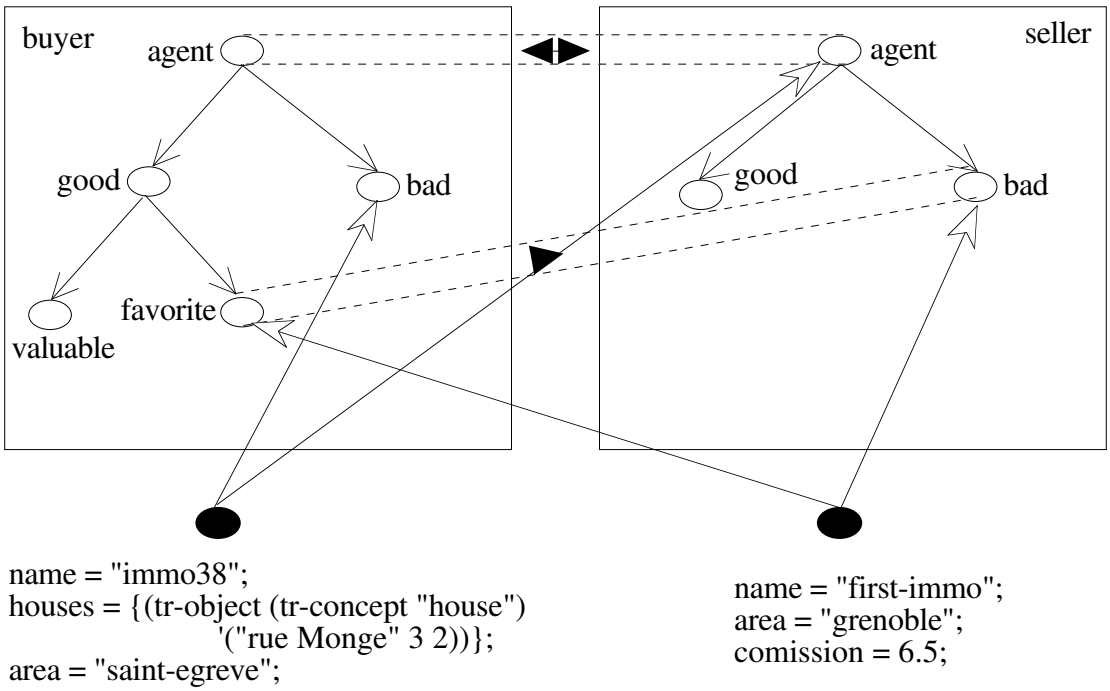


Figure 11: At the end of the complete example session, the resulting situation is presented here.

```
? (tr-classify (tr-object (tr-concept "agent") '("first-immo")))
= ((((tr-class (tr-conceptview (tr-concept "agent") "buyer") "favorite"))
    ())
   (((tr-class (tr-conceptview (tr-concept "agent") "seller") "bad"))
    ()))
? (tr-classify (tr-object (tr-concept "agent") '("immo38")))
= ((((tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad"))
    ())
   (((tr-class (tr-conceptview (tr-concept "agent") "seller") "agent")
     (tr-class (tr-conceptview (tr-concept "agent") "seller") "bad")
     (tr-class (tr-conceptview (tr-concept "agent") "seller") "good"))
    ()))
? (tr-attachable
    (tr-object (tr-concept "agent") '("immo38"))
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
= imp
? (tr-attachable
    (tr-object (tr-concept "agent") '("immo38"))
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "bad"))
= pos
? (tr-attachable
    (tr-object (tr-concept "agent") '("immo38"))
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "agent"))
= pos
? (tr-attachable
    (tr-object (tr-concept "agent") '("immo38"))
    (tr-class (tr-conceptview (tr-concept "agent") "seller") "good"))
= pos
? (tr-attachable
    (tr-object (tr-concept "agent") '("first-immo"))
    (tr-class (tr-conceptview (tr-concept "agent") "buyer") "good"))
= pos
```

Categorisation

Categorisation consists in inferring the possible positions, in a taxonomy, for a particular class definition. This capability is not provided yet.

Clustering

Clustering consists in organising a set of objects into one or several taxonomies. Clustering methods divide the set of objects into previously unknown classes whereby the main principle may be summarized as follows : the most similar objects are put in the same class and the less similar ones in distinct classes. A function of proximity is used to evaluate the likeness degree of pair of objects. In TROEPS, the clustering facility is based on a universal dissimilarity among the objects manipulated by the system, it can therefore cluster the objects of any concept and return the result as a taxonomy of intensionnally described classes. An environment to host various methods for automatic taxonomy construction, T-TREE, is included in TROEPS.

A detailled account of clustering is given here. However, it can be summarised as follows:

1. determine the concept to cluster and provide the weights for the fields on which the dissimilarity will be computed (see Screendump 48);
2. T-TREE will ask how to compute dissimilarity on objects of concepts which are the type of selected fields. This can be done either by using a particular viewpoint or by choosing to cluster this concept (see Screendump 49). In the second event, step 1 will be called again;
3. T-TREE returns the computed cluster, i.e. a new viewpoint with a class taxonomy (see Screendump 50).

For performing this, the system will, for each viewpoint, in a relevant order, compute the dissimilarity between each pair of objects of the concept. This will depend on the dissimilarity between their field values and the weights applied to field by the dissimilarity measure. Then, the system will aggregate objects into sets (the most similar objects first) and the sets will be aggregated into supersets (forming a taxonomy). Last, all these sets will be provided by class descriptions. This is the resulting taxonomy.

Dissimilarity

The aim of a proximity measure is to evaluate to what extent a pair of objects are alike (or unlike) with respect to the available information about them. Within an object model this information may be extracted either from object fields values or from the class taxonomy of the underlying concept (the mutual position of objects within the taxonomy).

In both cases the comparison of a pair of objects takes advantage of the fundamental principle of the object paradigm, which is the hierarchical organisation of the manipulated entities. Thus, the T-TREE built-in dissimilarity measure, called *topological*, is based on the fundamental principle of the object paradigm: organising the manipulated entities in a hierarchical manner. It relies on universally available relationships: sub-typing and class specialisation.

For a given pair of objects, the topological dissimilarity is computed with respect to their field values. First, a field-level function computes the differences between the respective pairs of field values. Then, an object-level function combines the values computed in this way into a single global result that caracterizes the compared objects.

The contribution of a field to the global dissimilarity value depends on the nature of the field domain. For abstract data types, i.e. primitive value domains, this contribution is computed on the hierarchical structure of meaningful sub-domains. In fact, given a pair of values, their dissimilarity is based on the relative *size* of their smallest common sub-domain. The size of a (sub-)domain is the number of domain values it includes. Thus the size of an integer interval [0 2] is 3 as there are three values, 0, 1, and 2. Surprisingly, when taken over an integer domain, the interval [1 1] is of size 1, whereas on a real one its size is 0. This rather complex way of computing the value dissimilarity has the advantage of being universally applicable on simple data types.

In fact, each basic data type connected to TROEPS (see Appendix 5) is provided with a function computing this value. Furthermore, the generic function, when applied on some standard types like integers, reals, etc. matches the well-known and mostly used distance functions like the arithmetic difference on integers.

For instance, given the three objects of the house concept presented in the following table:

objects \ fields	address	number	floor
house-1	"grenoble"	113	0
house-2	"grenoble"	21	2
house-3	"eybens"	13	4

The topological dissimilarity will compare them with respect to the values of a previously selected sub-set of fields. For instance, if one chooses to use the `address`, the `floor` and the `number` fields, whereby their relative importance (or weights) are 0.5, 0.2 and 0.3 respectively, the following computation can be done.

First, the elementary distance between `house-1` and `house-2` with respect to the `address` field is 0, since the field type, string, is of unordered (nominal) type family. Thus, only identical items lead to a 0 value for the difference whereas in other cases that value is 1. Furthermore, the address field differences score to a 1 for both the pairs (`house-1`, `house-3`) and (`house-2`, `house-3`). The remaining two fields are processed, in a similar way. The specific point with `floor` and `number` is that their types are numeric (quantitative) and discrete. so the arithmetic difference is taken and a normalisation is performed on it. Starting with the difference on `floor` for (`house-1`, `house-2`), one first computes their most specific common interval which is [0 2] and corrects it by 1 (the mean of the sizes of [0 0] and [2 2], both scoring to 1). The remaining value is divided by 4, the size of the whole domain which yields 0.5. This exactly corresponds to the normalized arithmetic difference on [0 4].

Thus, during clustering, the dissimilarity between two objects can be computed by aggregating the dissimilarities with regard to each field. Precisely, the dissimilarity between each field value is computed (by the previous method) and a weighted combination of each measure is computed (weights are provided through Screendump 48).

For instance, the weighted sum (weighted City block function) of the example above will compute the following values:

d(`house-1`, `house-2`) = 0.5 * 0 + 0.2 * 0.5 + 0.3 * 0.92 = 0.1 + 0.276 = 0.376,

d(`house-1`, `house-3`) = 0.5 * 1 + 0.2 * 1 + 0.3 * 1 = 0.5 + 0.2 + 0.3 = 1,

d(`house-2`, `house-3`) = 0.5 * 1 + 0.2 * 0,5 + 0.3 * 0.08 = 0.5 + 0.1 + 0.024 = 0.624.

The same principle is applied on object-valued fields: the likeness between objects taken as field values depends on their respective positions within the underlying taxonomy. Actualy, a taxonomy provides a set of classes which represent the admissible sub-domains of the whole object domain which is itself defined by the underlying concept. The dissimilarity is measured on the most specific common super-class of both objects. The number of the member objects is taken as class size. Thus, the most specific the common class, the most similar the member objects. Besides, the same computation procedure, i.e. corrections and normalization, applies on both objects and simple values.

For instance, on Figure 13, the following inter-object dissimilarities may be computed :

d(`house-1`, `house-2`) = 0.4,

d(`house-1`, `house-3`) = 1,

d(`house-2`, `house-3`) = 0.6.

The topological measure allows the various aspects of the concerned objects to be taken into account: the resemblance of objects under different viewpoints are evaluated separately (in function of the length of the shortest path between them in the taxonomy) and then aggregated to a single, usually weighted, mean. In both cases, i.e. primitive value domains and object sets, the result of the field-level dissimilarity function is real number between 0 and 1.

The above computation clearly requires the presence of at least one meaningful taxonomy over the concerned concept. In some cases, however, it is impossible to ensure the availability of classes, especially when the whole process is aimed at class discovery. Therefore, the comparison of objects remains only possible with respect to their field values. In such cases, T-TREE applies the same dissimilarity function as on initial objects, i.e. a weighted City block distance. It also applies clustering in order to better characterise the domains of the fields of the initial concept.

A technique of this kind may lead, when some circuits in object definitions occur, to a deadlock between dissimilarity computation tasks. In fact, it may happen that the dissimilarity of a pair of objects depends on the dissimilarity of another pair which in turn depends on the first dissimilarity. In such a case, a fixed point computation is applied in order to simultaneously evaluate the proximity of all mutually dependant pairs. The definitions of the dissimilarity on each pair is used to extract characterization functions which further compose a system of equations. The underlying vector function is repeatedly applied on an initial solution until a suitable approximation of the function fixed point is obtained. Its components represent approximations of the dissimilarities saught.

Clustering algorithms

Several algorithms of the Sequential Agglomerative Hierarchical Non-overlapping familly are currently available (complete, single — a.k.a. nearest neighbours —, average linkage). They can be characterised as ascending binary clustering algorithms (which progressively aggregate classes, starting from the instances, and only aggregate two classes in a row). The algorithms begin by considering each object as a class. Then, at each step, they aggregate into a new class the two classes which contains the nearest objects with regard to the dissimilarity measure. This new class contains the objects of the two previous ones which are not considered further. The algorithm stops when there is only one class left.

Based on the above values, a clustering algorithm will try to divide the set of objects into homogeneous sub-sets. I.e., object groups will be looked for such that the member objects score to low values of the pairwise dissimilarity whereas the pairs of objects in different groups

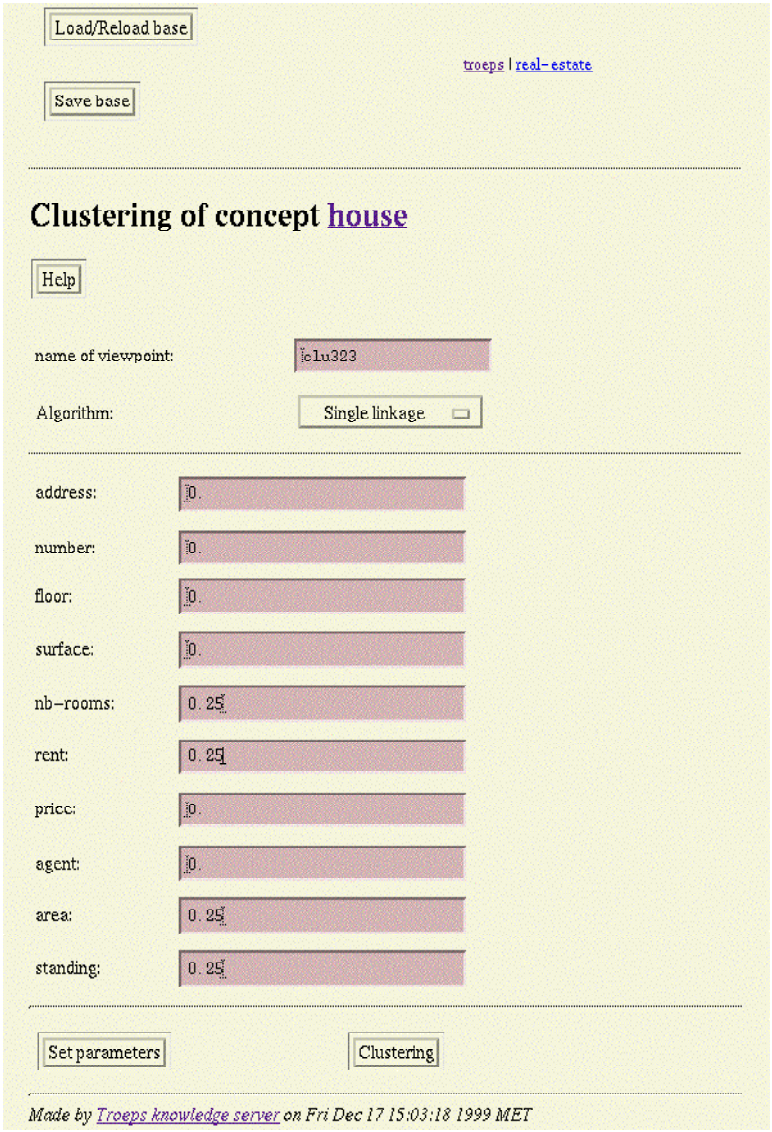
score to higher values. In the example, a sub-group will be created including `house-1` and `house-2` since their dissimilarity, 0.376, is substantially lower than the one of the pairs (`house-1`, `house-3`) and (`house-2`, `house-3`). Thus two classes will be generated `c1` and `c2`, having extensions {`house-1`, `house-2`, `house-3`} and {`house-1`, `house-2`} respectively. In other terms, `c2` is a sub-class of `c1`.

Generalisation

The result of the clustering phase is an inclusive hierarchy of sets of objects. In order to provide a description corresponding to these sets, a class description is inferred from each set of objects provided. The details of the generalisation mechanism provided are not described here. However, the application of the generalisation procedure on above example yields the following partial class descriptions:

```
<c1
   fields = {
      <address domain = {"eybens", "grenoble"};>,
      <number intervals = {[13 113]};>,
      <floor intervals = {[0 4]};>};>
<c2
   superclass = c1;
   fields = {
      <address domain = {"grenoble"};>,
      <number intervals = {[21 113]};>,
      <floor intervals = {[0 2]};>};>
```

Interface



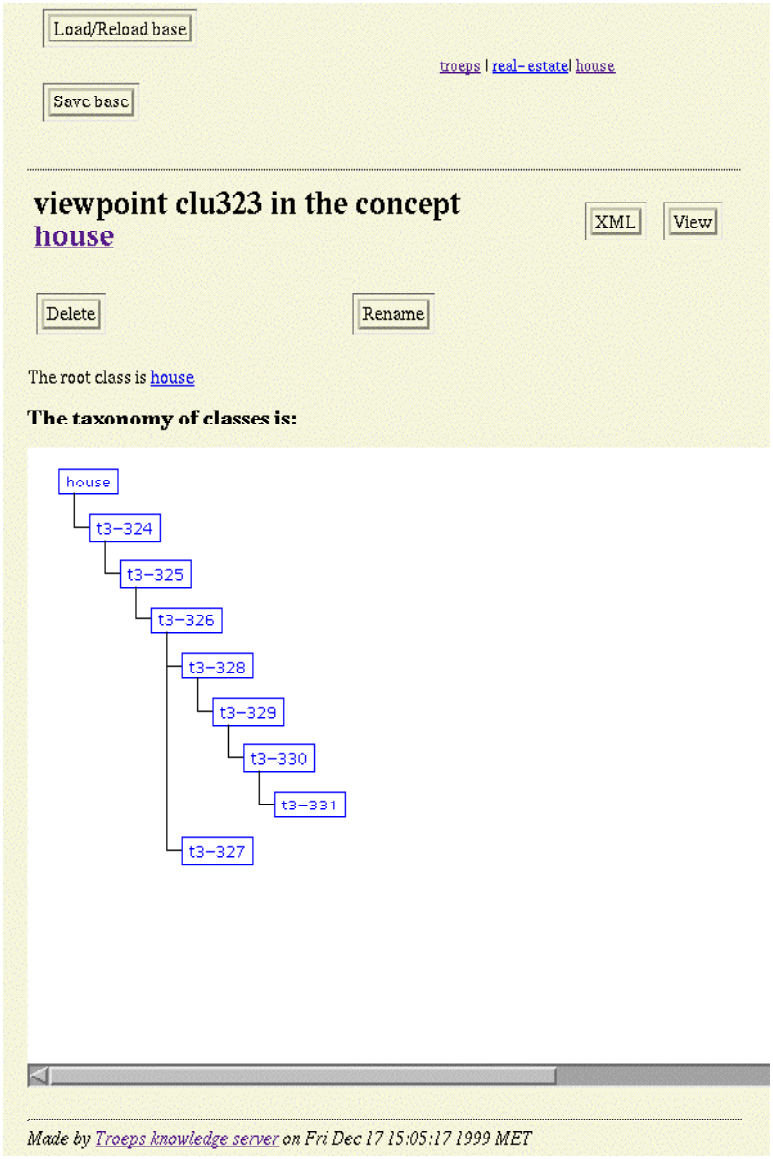
Screendump 48: Clustering initialisation.

Set parameters: .....NIY

Clustering:.....see Screendump 49, p. 104 or Screendump 50, p. 105

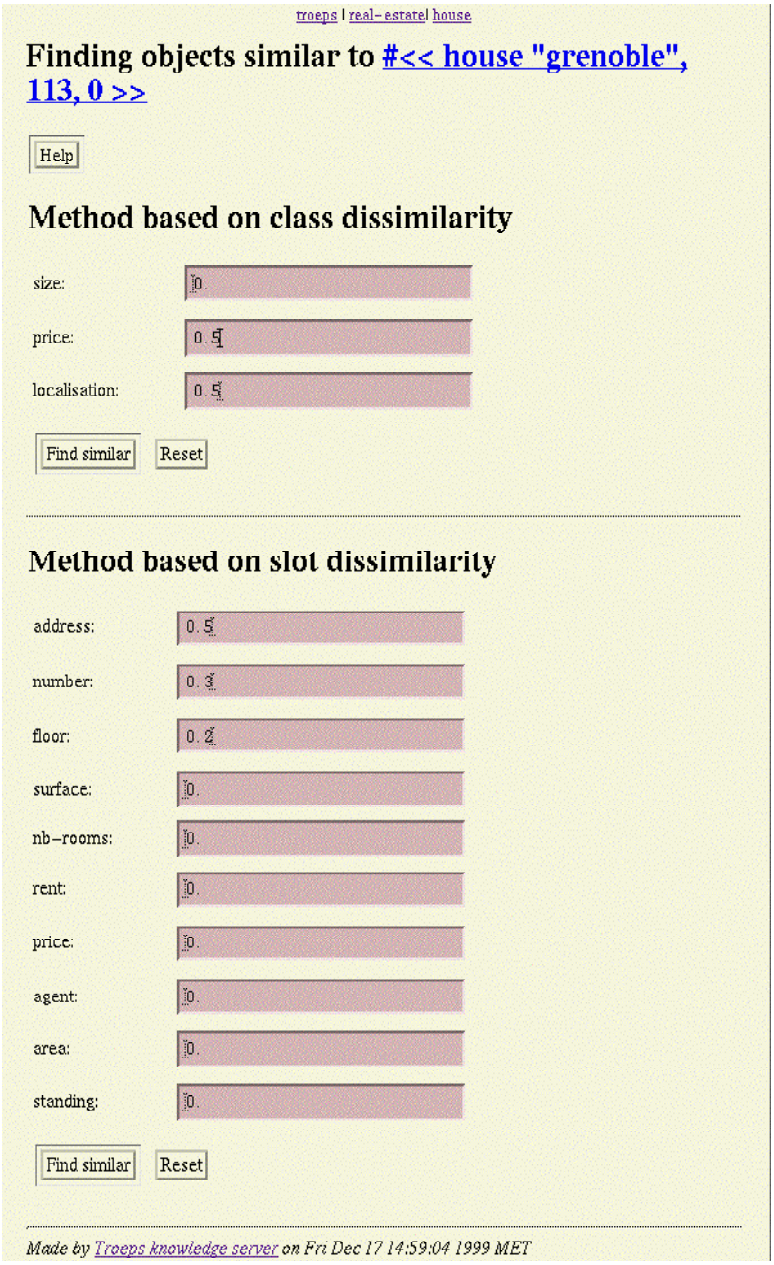
Screendump 49: Object dissimilarity completion.

Reset: ..... see Screendump 49
Validate: ..... see Screendump 48, p. 104 or Screendump 50, p. 105



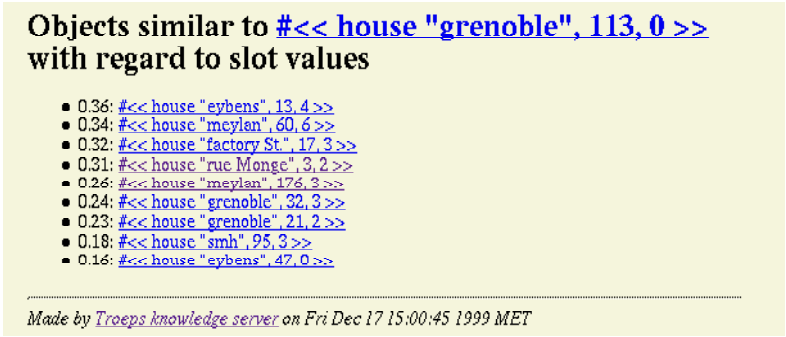
Screendump 50: Clustering result.

Edit: ..... see Screendump 12, p. 35
<class>: ..... see Screendump 14, p. 40



Screendump 51: Dissimilarity initialisation.

Find similar: ..... see Screendump 52, p. 106
<view>: ..... see Screendump 11, p. 34
Find similar: ..... see Screendump 52, p. 106
<field>: ..... see Screendump 8, p. 29



Screendump 52: Dissimilarity result.

<object>:..... see Screendump 31, p. 66

API

(**tr-clustering** *concept method create-p viewname slots*) function in [libtrt]

-> (), *concept* concept_{nak},*method* string_{nas}, *create-p* boolean, *viewname* $\text{string}_{nas,adkv}$, *slots* (($\text{conceptslot}_{naks}$. float_{bcf}) nal^*) nal

Builds a new taxonomy of classes for concept *concept* in a viewpoint called *viewname* (which is created if *create-p* is not `()`) by considering only the slots in *slots* with their respective weight comprised between `0.` and `1.`. The method used for clustering depends on the first element of the *method* argument (available methods are `"mv"`, `"al"`, `"sl"` and `"cl"`). This API function works perfectly for fields whose type is not another concept. In the opposite event, each such classslot must be created and the `clview` annotation set to the name of the concept view (of the domain concept) to use. This view must have been created. Either, it already has a taxonomy which is then used or it has class fields in its root class whose `clweight` annotation is set to a weight and `clalgo` whose value is the name of an algorithm as above and this recursively.

(**tr-find-slot-similar** *object slots*) function in [libtrt]

-> ((object . float)*), *object* object_{nao}, *slots* (($\text{conceptslot}_{naks}$. float_{bcf}) nal^*) nal

Compare the object *object* with all the other instances of its concept on the basis of the set *slots* of fields weighted by floats between 0. and 1. (whose sum is 1.) and returns their list ordered by higher similarity. This uses the weighted agglomeration of the dissimilarity between the concerned field values.

(**tr-find-view-similar** *object views*) function in [libtrt]

-> ((object . float)*), *object* object_{nao}, *views* (($\text{conceptview}_{nakv}$. float_{bcf}) nal^*) nal

Compare the object *object* with all the other instances of its concept on the basis of the set *views* of views weighted by floats between 0. and 1. (whose sum is 1.) and returns their list ordered by higher similarity. This uses the weighted topological dissimilarity over the concerned views.

Examples

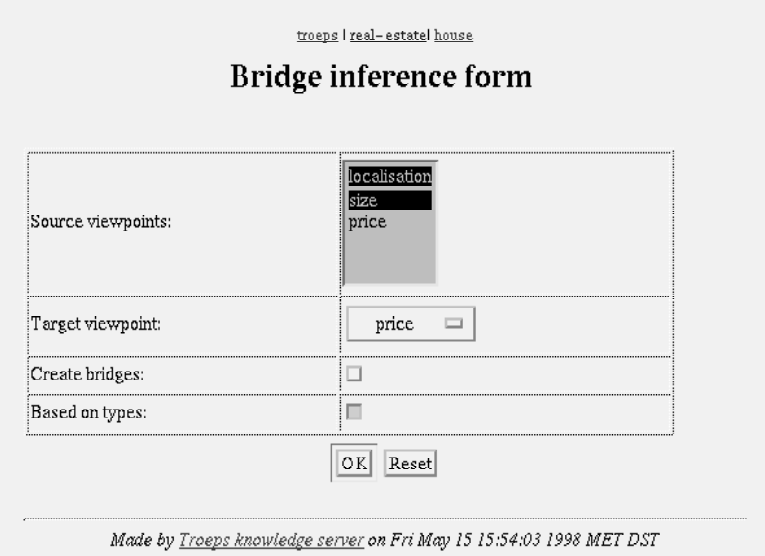
Bridge inference

Once taxonomies have been built and objects have been attached, it is very convenient to be able to compare them. To that extent, bridge inference algorithms are provided which produce bridges holding between the classes of the taxonomies.

The algorithms are used from a set of classes and a class and they look for all the bridges which involve subclasses of the former as source and subclasses of the latter as destination. The algorithms compare the classes and if all the “conjunction” of the source classes is smaller than the destination classes. However, it only returns the minimal bridges in the meaning provided in the Bridge section. So, the user can choose to implement a non minimal bridge. But any bridge not covered by a bridge returned by the algorithm is not valid.

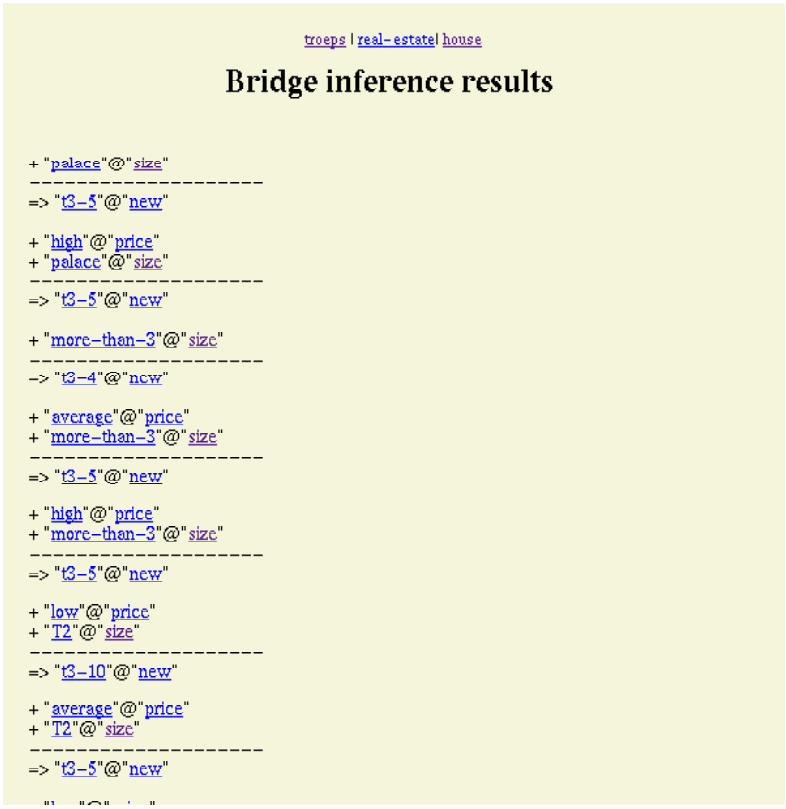
There are two versions of the algorithm depending on the predicates they rely on. A first version uses the members of the classes such that conjunction is intersection of these member sets and comparison is inclusion; the second version uses the types of the classes such that conjunction is type intersection and comparison is a sub-typing test.

Interface



Screendump 53: Bridge inference initialisation.

Infer bridges: ..... see Screendump 54, p. 108
Cancel:.....see Screendump 6, p. 23



Screendump 54: Bridge inference result.

API

(**tr-find-bridges-from-objects** *classes class create-p*) function in [libtrt]

-> { ((class*) . class)* }, *classes* class_{nas},*class* $\text{class}_{nac,ivc}$, *create-p* boolean

Looks for all the minimal possible bridges which have a subset of subclasses of *classes* as source and a subclass of *class* as destination. The possibility of a bridge is judged with regard to the set of objects members of each class. The bridges are created if *create-p* evaluates to true, their compositions are returned otherwise.

(**tr-find-bridges-from-types** *classes class create-p*) function in [libtrt]

-> { ((class*) . class)* }, *classes* class_{nas},*class* $\text{class}_{nac,ivc}$, *create-p* boolean

Looks for all the minimal possible bridges which have a subset of subclasses of *classes* as source and a subclass of *class* as destination. The possibility of a bridge is judged with regard to the type of each class. The bridges are created if *create-p* evaluates to true, their compositions are returned otherwise.

Examples

Complete example

Let the set of objects be the one contained in the real-estate knowledge base provided in the example directory. The set of 10 objets is given below with the relevant slot values:

#	address	number	floor	area	standing	nb-rooms	rent
1	factory St	17	3	grenoble	fair	4	6000.
2	meylan	176	3	meylan	fair	3	8000.
3	smh	95	3	smh	low	1	3000.
4	grenoble	113	0	grenoble	low	2	4000.
5	meylan	60	6	meylan	high	4	11000.
6	eybens	47	0	eybens	fair	2	7000.
7	eybens	13	4	eybens	fair	8	11000.
8	grenoble	21	2	grenoble	low	3	8000.
9	grenoble	32	3	grenoble	high	5	14000.
10	rue Monge	3	2	paris	fair	2	15000.

These object are first classified and attached to the most non ambiguous specialised class (i.e. the lowest class to which the objects can be attached while they cannot be attached to a non comparable class). For instance, the house #4 is attached to `other-area` while it can be attached to both `industrial` and `residential`. The result is given in the table below and in Figure 12. It is noteworthy that object 5, 9 and 10 are not attached to `high` under the `price` viewpoint because of a bridge stating that the high priced houses are palaces (that which these houses cannot pretend to be since they have at most 3 rooms). On the contrary, house #7 is attached to `high` because it can be attached to `palace`.

#	localisation	size	price	new
1	center	More than 3	average	t3-5
2	suburbs	T3	average	t3-5
3	suburbs	H	low	t3-10
4	other-area	T2	low	t3-10
5	suburbs	More than 3	house	t3-9

6	suburbs	T2	average	t3-5
7	suburbs	Palace	high	t3-5
8	other-area	T3	average	t3-10
9	center	More than 3	house	t3-9
10	center	T2	house	t3-5

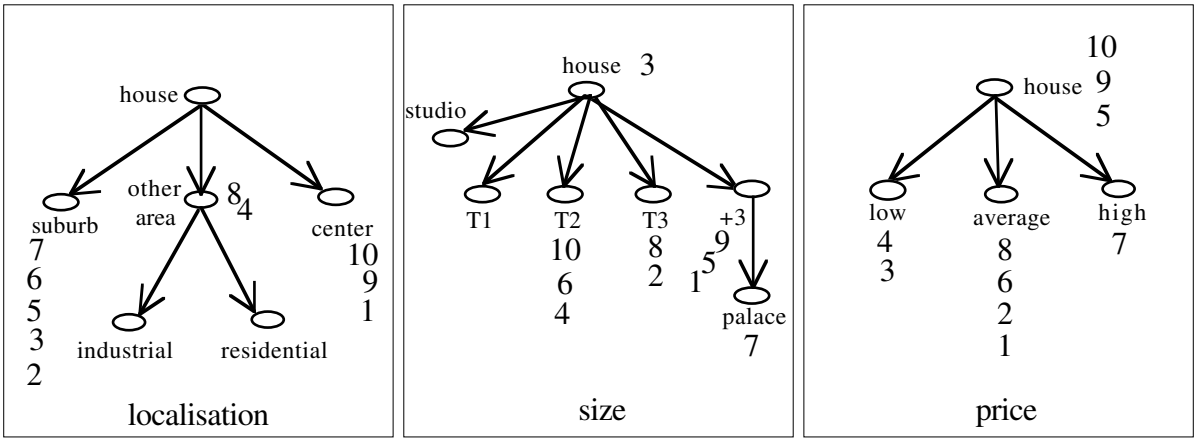


Figure 12: The various viewpoints and the attached objects after classification.

A new viewpoint is created by clustering on the slots `area`, `standing`, `nb-rooms` and `rent` with the same .25 weight. Interesting resulting classes are provided in Figure 16. It retains only the upper classes (the other ones have been discarded — that which, as like as renaming classes, is easily done with the graphic user interface). It can be noted that only one field (`standing`) allows to discriminate the classes. This is due to the importance of the dissimilarity between different nominal values. Overall, the classes t3-10 and t3-9 corresponds to two homogeneous groups (they can be independently discriminated by the `standing`, `rent` and `nb-rooms` fields.

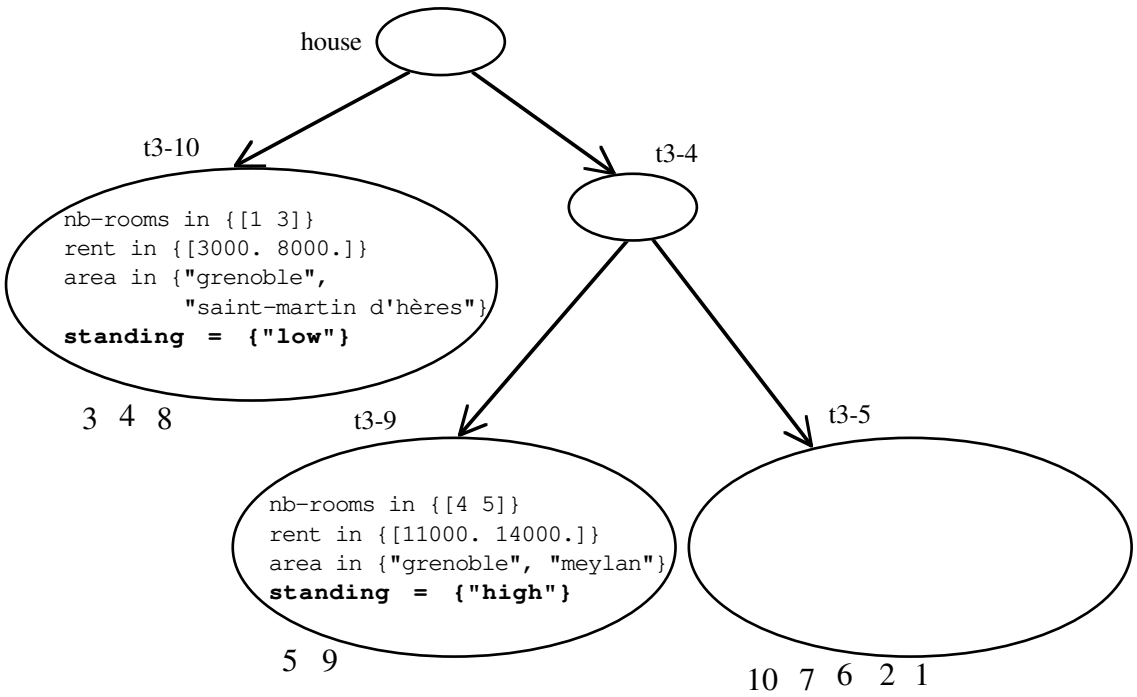


Figure 13: Interesting classes in the new viewpoint.

Last, it is possible to call the bridge inference methods in order to investigate the correlation between classes of different viewpoints. Figure 14 displays the result when using the inference from objects between viewpoints size, price and the newly created viewpoint.

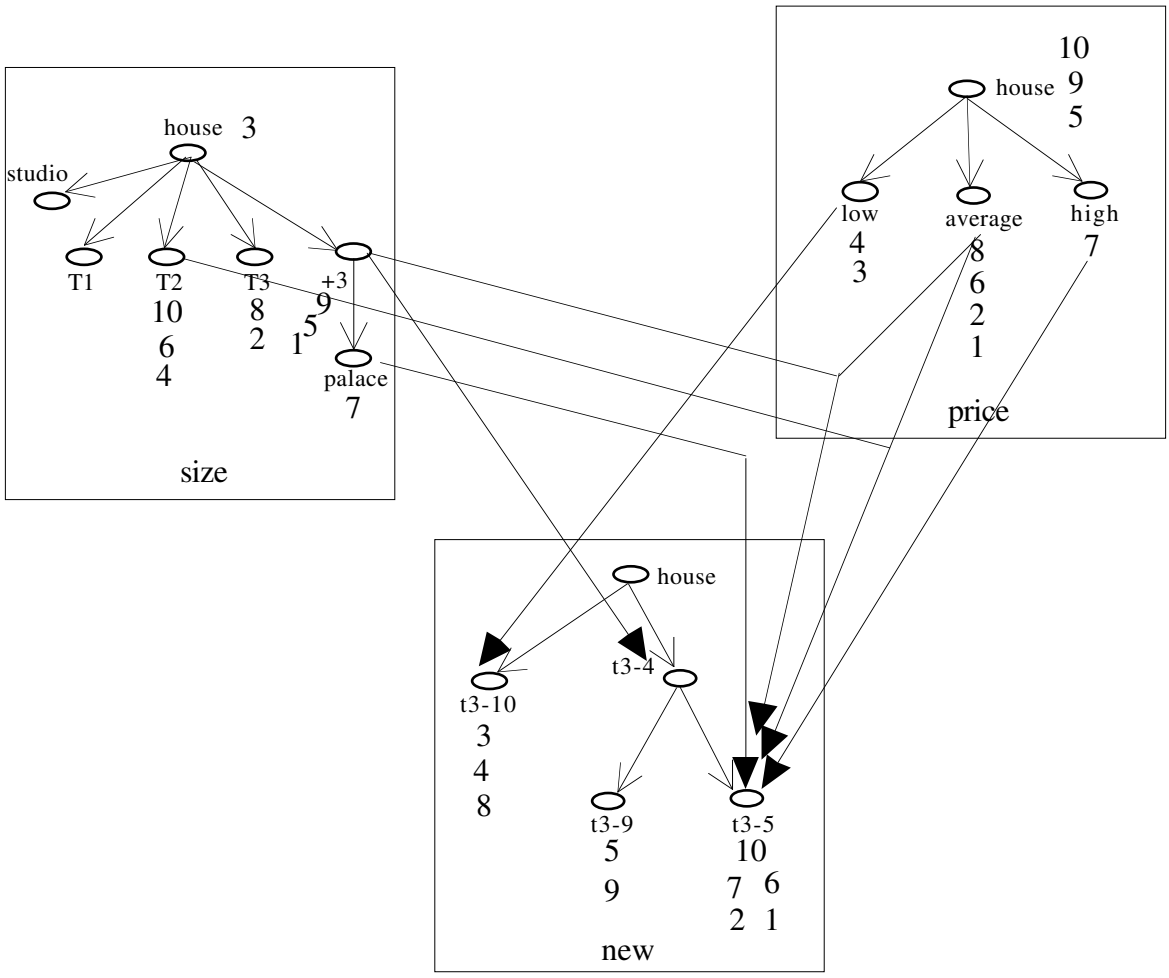


Figure 14: Bridge inference based on objects. Here are all the maximal bridges which do not target the root class.

Knowledge servers

One of the advantages brought by the creation of a knowledge base its the ability to browse through it. In TROEPS, this can be achieved very easily and widely through the use of the world wide web. On-line sample knowledge bases are available at the URL `http://co4.inrialpes.fr` which contains extensive material about TROEPS as an HTTP server.

Setting up your own server

TROEPS can be used as a knowledge base server accessible through HTTP. For that purpose, you must have a HTTP server running on one of your machines with the opportunity to add CGI scripts. Let us consider that a machine whose IP name is `myserv.mydom.domain` runs a HTTP server on port `80` and launches CGI scripts with the URL (Uniform Resource Locators) prefixed by `/cgi-bin`. Also assume that the TROEPS server will run on a machine (`itsmach.itsdom.domain`) which can be different from the HTTP server on the port `5555`.

In order to set up your Troeps server, you must:

1. put `$TRDIR/$PORTNAME/bin/hytropes` in the directory with CGI scripts.
2. ensure that your `LD_LIBRARY_PATH` must contains `${TRDIR}/${PORTNAME}/lib`.
3. Create a TALK file on the model of `htrp-config.t` containing instructions to be processed when the server is launched (for instance, loading the knowledge base).
4. Run, on `itsmach.itsdom.domain`, `trpserver` with the correct options:

```
$TRDIR/$PORTNAME/bin/trpserver \
-j http://co4.inrialpes.fr/java \   # non compulsory
-d http://co4.inrialpes.fr/help \   # non compulsory
-b http://myserv.mydom.domain:80/cgi-bin/hytropes \
-n itsmach.itsdom.domain \
-p 5555 \
htrp-config.t                              # non compulsory
```

5. Your base is now accessible from the following URL:

```
http://myserv.mydom.domain:80/cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/home-page
```

It is also possible to integrate the Java classes used for displaying the class hierarchies in your server by copying them (i.e. the `troeps-classes.zip` file) from the `co4.inrialpes.fr` server in a directory (say `/java` on your HTTP server) and changing the `-j` option in the line above to `http://myserv.mydom.domain:80/java`. The same thing can be done for help pages: they can be found on the same server (`help` directory) and can be set through the -d option to `http://myserv.mydom.domain/help`.

HTML output

Any TROEPS entity can be printed as a HTML (HyperText Mark-up Language) page. This can be used for browsing in a knowledge base in the absence of graphic user interface or for making the base available for others. Experiments have been carried out on generating the whole set of pages corresponding to a document and on generating on the fly the pages requested to the HTTP server.

Any TROEPS entity is provided with an URL that can be safely referred to by other web sites. When the URL is passed to the server through a client, the server will return the HTML page corresponding to the entity.

The URL is always prefixed by what corresponds to:

```
http://myserv.mydom.domain:80/cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/get/
```

and has an individual suffix as follows:

base	`<basename>/<basename>.html`
concept	`<basename>/<conceptname>/<conceptname>.html`
conceptslot	`<basename>/<conceptname>/s/<slotname>/<slotname>.html`
bridge	`<basename>/<conceptname>/b/<bridgename>/<bridgename>.html`
conceptview	`<basename>/<conceptname>/v/<viewname>/<viewname>.html`
class	`<basename>/<conceptname>/v/<viewname>/<classname>/<classname>.html`
classslot	`<basename>/<conceptname>/v/<viewname>/<classname>/<slotname>/<slotname>.html`
object	`<basename>/<conceptname>/o/<urlencodedstringobject>.html`

The objects are normalised with regard to their printable form:

```
#<<conceptname keyvalue1,… keyvaluen>>
```

This form is transformed into a string which, in turn, is transformed into a URL-encoded string. By URL-encoded is meant a string in which spaces are replaced by + and special characters are replaced by %xx such that xx is their hecadecimal ISO-Latin code.

For instance, the house concept corresponds to the URL:

```
real-estate/house/house.html
```

and the `#<<house "rue Monge", 3, 2>>` object corresponds to the URL:

```
real-estate/house/o/%23%3C%3Chouse+%22rue+Monge%22,+3,+2%3E%3E.html
```

Read-only and editable servers

The server can be set read-only or editable by setting a global variable `*edit-p*` in the `htrp-config.t` file. The `()` value means no edition while any other value enable the edition of the knowledge base. `()` is the default value.

Security

The access to the (either read-only or editable) server can be further refined through the basic security mechanisms of your HTTP server. Access control can be tied to particular pages, particular IP addresses or particular users (through passwords). As an example, it is possible to control the access to the edition, clustering and shutdown URL to the particular users in the edit group (identified by password given in the corresponding files). Here is the corresponding part of the `httpd.conf` file for the CERN 3.0 HTTP server:

```
#
#  Protection for Troeps files (for CERN server)
#

Protection EDIT-RIGHT {
   UserId nobody
   GroupId nogroup
   ServerId co4
   AuthType basic
   PasswdFile /myconfdir/passwd
   GroupFile /myconfdir/group
   GetMask edit
}
```

```
Protect /cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/shutdown/* EDIT-RIGHT
Protect /cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/edit/* EDIT-RIGHT
Protect /cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/lex-edit/* EDIT-RIGHT
Protect /cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/clustering/* EDIT-RIGHT
```

and of the `access.conf` file in the Apache 1.2 HTTP server:

```
# ****************************************
# *** Security access (for Apache server) **
# ****************************************

<Location       ~       "/cgi-bin/hytropes/itsmach.itsdom.domain:5555/TRP/(clustering|lex-
edit|edit|shutdown)/*">
   AllowOverride AuthConfig
   AuthName "Troeps"
   AuthType Basic
   AuthUserFile /myconfdir/passwd
   AuthGroupFile /myconfdir/group
   require group edit
</Location>
```

Editing knowledge bases through the server

All the screendumps appearing in the previous part of the manual have been created through the TROEPS HTTP interface. If some of the buttons in the screendumps do not show on the screen, the `*edit-p*` flag has certainly been assigned `()` (see above).

Customising the appearance

It is possible to control many parameters of the generation of knowledge bases. This is explained below. The kind of instruction given here can typically be placed in the `htrp-config.t` file of the server.

Adding HTML pages

The HTTP server can only generate pages from the knowledge it has. So, if it is able to display slot values and taxonomies, it is not able to create any picture or explanation of the notion used (even if it is connected to a lexicon). But, the users can add textual and pictorial information under the form of HTML pages which will be integrated in the pages generated on the fly whenever they are generated. The included pages can be arbitrarily complex (containing URL or images).

These pages reside in the machine where the server runs and are connected to the object through the annotations. The annotation values must be valid filenames. There are two annotations corresponding to two different places in the page:

- `prehtml` for the beginning of the generated page (under the title);
- `posthtml` for the end of the generated page (above the footer).

Thus, once the pages have been designed and before the server is launched, each page must be attached to its object (which can be a TROEPS object or a concept or a bridge: any TROEPS notion) by setting the corresponding annotation to the filename of the HTML page. For instance,

```
(tr-set-annotation
   (tr-find-object (tr-find-concept "house") (list "rue Monge" 2 3))
   "posthtml"
   #f"/home/bases/real-estate/html-pages/map.html")
```

will add to the display of the object at the end of the page, the content of the file given as value of the annotation (here `map.html`).

This feature does not support multi-linguism (i.e. the HTML can be generated in any language you want — provided that the messages have been defined — but these files are unique).

Linking to other bases

The added pages can be arbitrarily complex. They can contain text and pictures. They an also contain URL referring to static pages available somewhere in the WWW or dynamic pages such as those generated on the fly by TROEPS.

Modifying object pages

If the display of the objects of a particular concept must be improved, it is possible to do so. For that purpose, the display must be programmed in a function having the object and a boolean as arguments and printing a HTML page on the standard output. This function is tied to the base through an annotation (funhtml) attached to the concept. Thus, the instruction:

```
(tr-set-annotation (tr-find-concept "house") "funhtml" #'my-print-house)
```

tells the system that the instances of house have to be displayed through the function `my-print-house` instead of the standard procedure.

This function can be used in order to generate the page depending on the object which can be manipulated through the TROEPS API. For instance, it is possible to embed in the page URLs which are built from the content of the knowledge base but correspond to queries to remote resources (such as a site providing e-mails of individuals).

Modifying object pages also allows to embed applets into the display since from the object, all the data necessary for passing as applet parameters are available. The call to the applet can be used in order to show some features of the object through specialised editors (e.g. a graph editor) which will run on the client side.

Adding functions

The TROEPS HTTP server uses a set of URL in order to identify objects and the operations on these objects (e.g. editing, classifying). It is however possible to extend the set of URL understood by the server and to provide the functions which correspond to these URL.

The advantage of that technique is that the functions are called in the TROEPS environment and thus it can access the knowledge base through the TROEPS API and compute the result or even modify the knowledge base. It is possible to build a complete set of pages on that model (and the TROEPS editor was first build that way).

In order to do so, the `manage-url-hook-names` dynamic variable contains a list of names of functions to apply when the URL is not recognised by the regular server. Each function will be called with the URL remaining after `hytropes` as argument and must return an integer (0 if the URL has not been recognised and something else otherwise). Typically these hooks will first recognise their own keyword after `hytropes/`.

The input can be decoded with the help of a few functions provided by TROEPS:
`troeps.http.urldecode`: decode the URL between two positions;
`troeps.http.file-object`: transforms a piece of this URL into the corresponding TROEPS object.

The behaviour of the function must be to print HTML data on standard output. If `manage-url-hook-name` is empty, or if each function returns 0, the `nrk-reply` function is called notifying the non recognition through HTTP.

Example

As an example, here is a typical `htrp-config.t` file illustrating all the features presented above:

```
;;; This is the configuration file for a Troeps server
;;; ================================================================

;;; Required information

(setf (global *dir-prefix*) "")
(setf (global *edit-p*) t)

;;; ================================================================
;;; Initialising

(tr-init)
(setf (languages) '(french english))

;;; ================================================================
;;; Loading sample base

;;; loading the knowledge base
(tr-load-base #f"real-estate.bdf")

;;; ================================================================
;;; Adding HTML page annotations

(tr-set-annotation (tr-current-base) "prehtml" #f"real-estate.html")
(tr-set-annotation (tr-object (tr-concept "house") '("rue Monge" 2 3)
  #f"house/rue-monge-2-3.html")

;;; ================================================================
;;; loading additional talk functions

(load-file #f"real-estate.t")

;;; initialisation of these functions
;;; (like computing a map of the area from Troeps structures)

(re-init)

;;; ================================================================
;;; special printing function for displaying the instances of concept person

(defun re-display-person (object edit)
   ;; re-use the usual description
   (troeps.http.print-html-description object edit)
   ;; Add an inlined button for fetching the e-mail of the person
   (printf "<LI><FORM METHOD=POST\n")
      (printf "ACTION=\"http://www.four11.com/cgi-bin/SledMain?")
      (printf "FS,234,2,377FB00,3F70FA11\">\n")
      (printf "  <INPUT TYPE=hidden NAME=\"FirstName\" VALUE=\"%s\">\n"
         (tr-get-value object "firstname"))
      (printf "  <INPUT TYPE=hidden NAME=\"LastName\" VALUE=\"%s\">\n"
         (tr-get-value object "name"))
      (printf "  <INPUT TYPE=submit NAME=\"Search\" VALUE=\"Find email\">\n")
      (printf "</FORM></UL>\n")
   ;; Add an Applet (using the Troeps tree display applet)
   (printf "<APPLET CODE=\"TreeDisplayApplet.class\"\n")
   (printf "        CODEBASE=\"%s\"\n" (global *url-java*))
   (printf "        ARCHIVE=\"troeps-classes.zip\"\n")
   (printf "        WIDTH=600 HEIGHT=500>\n"))
```

```
     ;; Get the parameters: the set of houses owned by the person
     (printf "  <PARAM NAME=tree VALUE=\"(%s" (tr-get-value object "firstname"))
     (for ((h in: (tr-get-value object "houses")))
       (printf " (%s)" (tr-get-value object "address")))
     (printf ")\">\n")))
  (printf "</APPLET>\n")
  ;; Add buttons for calling other pages (printing picture)
  (troeps.http.display-button
     (catenate *url-prefix* "/re/print-picture" (troeps.http.refname object))
     "Afficher doc")
)

;;; ============================================================
;;; Adding display annotations (after the definition of functions)

;; Tell the system to use the above function for printing person instances
(tr-set-annotation (tr-concept "person") "funhtml" #'re-display-person)

;;; ============================================================
;;; Menu for special functions
;;; (like printing the map on demand and getting the instance from a clic)

(defun re-manage-url-hook (url)
   (let* ((begin (i1+ (char-index #\/ url 1)))
          (end (char-index #\/ url begin))
          (keyword (troeps.http.urldecode url begin (i1- end))))
      ;; Recognise the re prefix
      (if (string= keyword "re")
          (progn
             (setf begin (i1+ end))
             (setf end (char-index #\/ url begin))
             (setf keyword (troeps.http.urldecode url begin (i1- end)))
             (cond
                ;; Recognise the print-map action
                ((string= keyword "print-map")
                   (re-print-map))
                ;; Recognise the print-picture action
                ((string= keyword "print-picture")
                   ;; Get the object corresponding to the URL and print it
                   (re-print-picture (troeps.http.file-object (substring url (i1+ end)))))
                (t 0)))
          0)))

;; Tell the system to use the above function for recognising re-prefixed urls
(pushf (dynamic manage-url-hook-names) 're-manage-url-hook)

;;; ============================================================
 (printf ";;; real estate initialised\n")
```

API

(tr-save-base-as-html *base dirbase*) function in [libtrh]

-> boolean,*base* base_{nab}, *dirbase* $\text{string}_{\text{nas}}$

Prints the knowledge base *base* as a HTML document. The result is printed in a directory whose name is that of the base in the directory *dirbase*. All the URL used in the output are relative. This means that the files are stored under the *dirbase* directory but the URL that they contains do not refer to this directory (it is only relative to the current directory). Note that all the dynamic capabilities of the HTTP server are not available under that mode (no TROEPS filtering, no editing and so on). Returns `t` if no error has been detected, signals an error otherwise.

(troeps.http.file-object *url*) function in [libtrh]

-> entity, *url* string

Returns the TROEPS entity corresponding to the string *url* interpreted as a URL as generated by the function `link`. If no object corresponds to the string, an error message is returned on standard output under the form of a HTML page.

(**troeps.http.urldecode** *url begin end*) function in [libtrh]

-> string, *url* string, *begin* integer, *end* integer

Returns a string corresponding to the substring between position *begin* and *end* in the entry string *url*. The results is decoded according to the HTTP encoding rules.

(**troeps.http.link** *entity*) function in [libtrh]

-> string, *entity* entity

Returns the string corresponding to a HTML link to the TROEPS entity *entity* (it will be decoded through `file-object`).

(**nrk-reply** *keyword socket*) function in [libtrh]

-> boolean, *keyword* string, *socket* stream

Returns on the socket *socket* a HTML page signaling that the string *keyword* is not understood by the HTTP server.

(global ***edit-p***) global-variable in [libtrh]

-> boolean

Indicates if the TROEPS server is read-only (`()` value) or writable (any other value).

(global ***revision-p***) global-variable in [libtrh]

-> boolean

Indicates if the TROEPS server is will trigger the revision mechanism when an error is encountered or not (`()` value).

Concurrent editing with Co4

Co_4 is a framework allowing to relate several TROEPS knowledge bases in order to build consensual knowledge bases. To that extent, the Co_4 protocol deals with the registration and submission from a knowledge base to another (called group base). The group base processes automatically all the queries from its subscribers.

For more information on Co_4, see `http://co4.inrialpes.fr/docs/co4-manual.html` which provides pointers to Co_4 articles, documentation and software. The feature considered here is the connection of TROEPS to Co_4, it cannot be read in isolation from the Co_4 documentation. In the following, it is considered that :

— TROEPS is installed in `$TRDIR`;

— Co_4 is installed in `$CO4DIR` and has been compiled for an ANS (Agent name server) `$ANSURL`.

A ready to use TROEPS-Co_4 connection is provided in the `$TRDIR/examples/co4` directory. It can be recompiled for your installation by simply editing the `makefile` for setting the `TRPDIR` and `CO4DIR` variables and typing:

```
$ make
```

in that directory. This will put a new binary called `co4trpserver` in the `$TRDIR/$PORTNAME/bin` directory.

Like for the TROEPS server, in order to set up your new CO_4-handled TROEPS server, you must (* means that it has been already done for installing the TROEPS knowledge servers):

1*. put `$TRDIR/$PORTNAME/bin/hytropes` in the directory for CGI scripts.
2*. ensure that your `LD_LIBRARY_PATH` contains `${TRDIR}/${PORTNAME}/lib`.
3. Create a TALK file on the model of `htrp-config.t` containing instructions to be processed when the server is launched (for instance, loading the knowledge base).
4. Run `co4trpserver`, on `itsmach.itsdom.domain`, with the correct options:

```
$TRDIR/$PORTNAME/bin/co4trpserver \
   -c <name> $ANSURL <co4port> <troepsport>\
   -j http://co4.inrialpes.fr/java \ # non compulsory
   -d http://co4.inrialpes.fr/help \ # non compulsory
   htrp-config.t                     # non compulsory
```

Your base is then accessible from the following URL:

```
http://myserv.mydom.domain:80/cgi-bin/hytropes/itsmach.itsdom.domain:5555/CO4/home-page
```

There are some differences between a simple TROEPS knowledge server and a CO_4-handled one:

— the TROEPS editor appears in a frame of CO_4;
— the knowledge bases are (preferebly) stored in the ANS server rather than the local disc;
— the `htrp-config.t` can contain special instructions for the `co4trpserver` (through the `*bases-to-load*` variable which must be set to a list of knowledge base description files).

It is noteworthy that the CO_4-handled TROEPS knowledge servers can be shutdown and waken up. At shutdown time, the system stores in the ANS the content and the parameters of the knowledge bases. During the time it is down, the messages addressed to the server are stored in a mailbox and at wake-up time, the content and the parameters of the knowledge base are restored and all the messages are visible from the interface.

The shutdown is processed from the HTTP client but the wake-up must occur at the shell top level (so far). It is achieved by using the `-w` option of `co4trpserver`:

```
$TRDIR/$PORTNAME/bin/co4trpserver \
   -w <name> $ANSURL <co4port> <troepsport>\
   -j http://co4.inrialpes.fr/java \ # non compulsory
   -d http://co4.inrialpes.fr/help \ # non compulsory
   htrp-config.t                     # non compulsory
```

Appendix 1. How to get, install and run TROEPS

Requirements

TROEPS is being developed on Sun SparcStations under SunOS 5.4 (a.k.a. Solaris). It runs and has been tested under SunOS 4.3 and Windows-NT; it can be ported to other platforms either by INRIA or by yourself by recompiling the sources. It can be used as a library linkable to other programs.

TROEPS has been developed with the standard Unix utility Lex, YACC (now replaced by FLex and Bison) and the Ilog TALK LISP compiler. It is thus made of a set of C programs that should be compilable with any C compiler as soon as Ilog TALK libraries are available. The knowledge servers use a small set of Java applet and should be used on any kind of platform on which it is possible to find a HTTP client.

Ilog TALK compiler and libraries are available from Ilog SA, 9 rue de Verdun, BP 85, 94253 Gentilly, France.

Getting TROEPS and its documentation

TROEPS and its documentation are available for public distribution under the term of the COPYRIGHT file provided with the distribution. They can be retrieved through anonymous ftp (file transfer protocol) on the machine `ftp.inrialpes.fr` under the names:

```
/pub/sherpa/rapports/troeps-manual.ps.gz
/pub/sherpa/logiciels/troeps/current.tar.gz
```

Our HTTP server provides various information (among which the latest release and the manual on-line). The URL are:

```
http://www.inrialpes.fr/sherpa/
http://co4.inrialpes.fr/docs/troeps-manual.html
```

Do not hesitate to browse anywhere on our site (beginning with `http://co4.inrialpes.fr`) a lot of information about new developments is available there.

Installing TROEPS

TROEPS can be installed anywhere in your system provided that you follow the instructions below. Assume that TROEPS is installed in the directory `TRDIR`. The files can be obtained by:

```
$ gunzip troeps-manual.ps.gz
$ gunzip current.tar.gz | tar xf -
```

Let assume that `TRDIR` is stored in a variable `$TRDIR` and that there is a `$PORTNAME` variable. The content of the tar-file consists in the following files and directories:

`COPYRIGHT`	the INRIA Copyright notice;
`emacs`	a simple but useful TROEPS mode for emacs;
`examples`	contains some sample knowledge bases (useful for building your own);
`examples/cmain`	is a sample C program calling TROEPS;
`help`	the help HTML files for the interface;
`include`	contains the set of .h files for using TROEPS from C;

`java`	the JAVA applets for the interface;
`sparc_4_3.0`	contains C sources and libraries for SunOS 4.3 port;
`sparc_5_gnu`	contains C sources and libraries for SunOS 5.4 (Solaris) port with the gnu compiler;
`sparc_5_4.0`	contains C sources and libraries for SunOS 5.4 (Solaris) port;
`talk`	contains the TALK sources of TROEPS;
`csrc`	contains the C sources for analyser and hytropes cgi-script;
`tests`	contains the test files for TROEPS;
`trp.ws`	is the workspace description file for using TROEPS under TALK;
`trt.ws`	is the workspace description file for T-TREE;
`trr.ws`	is the workspace description file for revision
`htrp.ws`	is the workspace description file for using hypertext extensions.

Running TROEPS under the TALK interpreter

In order to use it, it is only required that the installation directory be in the path. This can be achieved by the following assertion:

```
(pushf (workspace-path) #f"$TRDIR")
(in-workspaces '(trpws))
```

which can be typed either in your `.talkrc` file, in another file that you load or an italk tty input. The following sample session shows a way to use TROEPS at the interactive toplevel of Ilog TALK:

```
$ italk
;;; ILOG Talk version 3.3 [sparc_5_gnu]
;;; Copyright (c) 1989-1997 by ILOG
;;; Write to talk-support@ilog.com for help
;;; Image saved on Fri Jul  4 19:26:59 1997
;;; Connecting to Emacs server...connected.
;;; Loading workspace `talk': Ilog Talk kernel workspace
;;; Loading workspace `trp': Troeps development workspace
;;; Loading workspace `trt': Troeps taxonomy building tool workspace
;;; Loading workspace `htrp': Troeps knowledge server development workspace
;;; Loading workspace `divers': executables and utility files workspace
;;; Welcome to ILOG Talk.
;;; Type (exit) to leave Talk.
? (load-program-unit 'libtrl)
;;; Loading library `libtrl': Troeps textual interface
;;; Loading run-time environment of `libtrl': (libtrp)
;;; Loading library `libtrp': The core Troeps library
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/lib/libtrp.so"
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/lib/libtrl.so"
= libtrl
? (load-program-unit 'trtdum)
;;; Loading module `trtdum': Troeps dummy basic type descriptions
;;; Loading object file "/home/lotse/sherpa/TROPES.0/src/sparc_5_gnu/o/trtdum.o"
= trtdum
? (tr-init)
= t
? (setf (languages) '(french english))
= (french english)
? (tr-load-base #f"examples/real-estate.bdf")
= t
? (setf co (tr-find-concept "house"))
= (tr-concept "house")
? (setf ob (tr-find-object co '("rue Monge" 3 2)))
```

```
= (tr-object (tr-concept "house") '("rue Monge" 3 2 ))
? (tr-get-value ob "surface")
= 45.
? (exit)
$
```

Running TROEPS as a WWW server

See the “Knowledge server” part.

Editing knowledge bases under emacs

A simple TROEPS mode is available under emacs (emacs 19 or above) when you add the following line in your `.emacs`:

```
(load-file "$TRDIR/emacs/troeps-emacs.elc")
```

The mode is triggered by files with `.bdf` and `.idf` extensions. It provides a very useful “fontification” of knowledge bases (colouring or changing fonts of keywords, strings... on the fly) and a pop-up menu allowing to insert a new frame (new concept, new bridge...) in a base.

Recompiling the TROEPS libraries

This feature is not available yet (except if you do it under the TALK environment in which case you should refer to the Ilog TALK manual).

Using the TROEPS libraries

TROEPS extension is programmed in Ilog TALK. However, it can be used from an alien language (C, C++, etc.). For that purpose, TROEPS is supplied with several libraries (`libtrp`, `libtrl`, `libtrh`, `libtrr` and `libtrt`) along with the corresponding include file (`trp.h`, `trl.h`) which enables its use from these languages. Some slight changes with regard to the manual must be observed:

- it is not possible to use macros (?), setf-able forms (this can be changed soon);
- the “-” in the names are replaced by “_”;

An example of C file using TROEPS is the following (which can be found under `$TRDIR/examples/cmain`):

```
#include <stdio.h>
#include <ctype.h>
#include <trp.h>
#include <trl.h>

#if defined(__STDC__) || defined(__cplusplus)
extern int IltInit(int argc, char** argv, char** envp);
#else
extern int IltInit();
#endif

#if defined(__STDC__) || defined(__cplusplus)
int main(int argc, char** argv, char** envp)
#else
int main(argc, argv, envp)
int argc; char **argv, envp;
#endif
{
  void *ob, *co;
  double *surf;

  void** vector_concept_views;

  IltInit(argc, argv, envp);
```

```
  tr_init();
  tr_set_language(TR_STR("french"));
  tr_load_base(TR_FILENAME("../real-estate.bdf"));
  co = tr_find_concept(TR_STR("house"));
  ob = tr_find_object(co, TR_CONS( TR_STR("rue Monge"),
                                   TR_CONS( TR_INT(3),
                                            TR_LIST( TR_INT(2)))));
  vector_concept_views = TR_VECTOR(0);
  surf = tr_get_value(ob, TR_STR("surface"), vector_concept_views);

  printf("%.2f is the value of the surface of %s.\n",
         *surf, ilt_string(ob));
  return(0);
}
```

It can be compiled through the following `Makefile`:

```
ILTINC=${ILT_DIR}/talk/common

# edit this variable for computing
TRPDIR=/home/lotse/sherpa/TROEPS.0/src
INC=${TRPDIR}/include
LDDIR=${TRPDIR}/${PORTNAME}/lib
MODULO=${TRPDIR}/${PORTNAME}/o/trtdum.o
TRELAB=${TRPDIR}/${PORTNAME}/o/trelab.o
CLIBS= -lnsl -lm -ldl

all: once Makefile

${PORTNAME}/o/pgm.o: ${INC}/trp.h ${INC}/trl.h pgm.c Makefile
        cc -g -c -PIC  pgm.c -I$(ILTINC) -I$(INC) -o ${PORTNAME}/o/pgm.o

${PORTNAME}/bin/pgm: ${PORTNAME}/o/pgm.o ${TRELAB} $(MODULO)  Makefile
        cc -PIC -o ${PORTNAME}/bin/pgm \
           -L$(LDDIR) -L${ILT_DIR}/lib/${PORTNAME} \
           ${PORTNAME}/o/pgm.o ${TRELAB} $(MODULO) \
           -ltrp -ltrl -ltrt -ltrh -l trr -liltcrt -liltrt -lilteval $(CLIBS)

once: ${PORTNAME}/bin/pgm
   ${PORTNAME}/bin/pgm
```

The elaboration of the program takes a while for saving an image of the resulting state. You can now call your program by `${PORTNAME}/bin/pgm`.

Reporting bugs/getting advice

This can be done preferably through mail to `troeps@inrialpes.fr`.

Appendix 2. Textual format

This Appendix describes both the current version and further improvements of the TROEPS textual format. A good idea is to follow the syntax used in the `real-estate.bdf` example which is provided with TROEPS. Note that many changes have occurred (the whole loader have been rewritten between version 1.0 and 1.1) and complies now with the specifications. So your old bases will not work anymore. The main differences are the following:

- bridges have a name;
- class specification are separated by the '@' symbol;
- inference descriptors have been added;
- the `lexicons = {};` syntactic phrase is required;
- the `instances = {};` syntactic phrase is not required anymore;
- the `facettes = {};` syntactic phrase is not required anymore;
- the `methods = {};` syntactic phrase is required (see below);

Principles

In TROEPS, the language is one mean (among others) and not an end. The system is defined through a set of basic notions which are accessible through TROEPS API. However, for reasons of textual dumping of a knowledge base into a file, of quick acquisition of data from another source and of global creation by a skilled user, TROEPS provides a textual format for defining knowledge bases.

Since it is not primitive, this format is subject to change but the TROEPS team warrants that, from one version of TROEPS to another, it will be able to translate from one format to another. The format is divided into a *scheme* format, used to describe concepts, viewpoints, classes, fields and bridges, and an *instance* format used for describing objects.

The base description file

A base description file contains the description of a knowledge base. Its name is terminated by a ".bdf" extension. It refers to the other knowledge bases (.bdf), lexicon (.ldf), annotation files (.adf), TALK files (.t) which have to be loaded, describe types and concepts to be declared inside the knowledge base and instance files (.idf) that must be loaded after these declarations.

Concepts in a knowledge base can only refer to concepts and types which have been declared either in the imported knowledge bases or in the present knowledge base. Of course, there cannot be any circularity between the references (but through link fields).

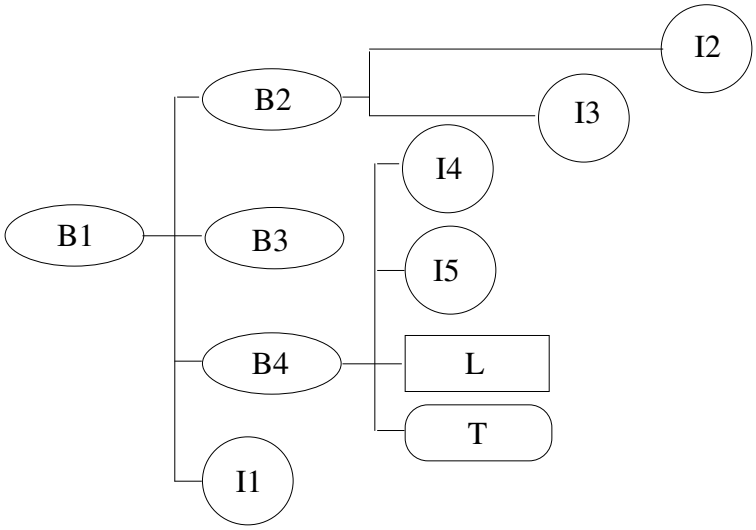


Figure 15: The knowledge base B1 contains the bases B2, B3 and B4 plus the instance file I1.

Usually, an application is made of several of these knowledge bases, one by concept to be declared ($B_{k1},\ldots B_{kn}$), several ones which are built by referring to these concepts ($B_{k'1},\ldots B_{k'm}$) and one application knowledge base (B) which gathers all these other bases and declare the instance files (see Figure 15).

Syntactic shape

TROEPS textual format is parsed with the help of a program generated by YACC, but the grammar given here is to be understood as a standard for the syntax and not as the actual implementation of it. Here is a small typical base description (boldfaced names are keywords). The syntax itself is introduced afterward.

```
<base-name
   required-bases = { base-name,… };
   types = { type-desc,… };
   viewpoints = { view-name,… };
   concepts =
      {
         <concept-name
            key = {
                  <keyslot-name
                     type = concept or type;
                     nature = propriete or composant or lien;
                     constructor = un or liste or ensemble;
                  >,
                  …
               };
            fields =
               {
                  <field-name
                     type = concept or type;
                     nature = propriete or composant or lien;
                     constructor = un or liste or ensemble;
                  >,
                  …
               };
            viewpoints =
               {
                  <conceptview-name
                     root-class =
                        <class-name
                           fields =
                              {
```

```
                        <field-name descriptor = desc-value;… >,
                        …
                     };
               >;
            classes =
               {
                  <class-name
                     super-class = class-name;
                     fields =
                        {
                           <field-name descriptor = desc-value;… >,
                           …
                        };
                  >,
                  …
               };
            bridges =
               {
                  <  bridge-name
                     source = { class-name@conceptview-name,… };
                     destination = class-name@conceptview-name;
                  >,
                  …
               };
         >,
         …
      };
   methods = { method-signature,… };
   instance-files = { file-name,… };
   lexicons = { file-name,… };
   talk-modules = { file-name,… };
>
```

Instance files

On loading a knowledge base from a particular file, the system will attempt to load an instance file with the same filename but ".idf" as extension. It will then load the files corresponding to the instance-file keyword. Instance files contain instance descriptions under the following format.

```
{
<concept-name
   is-in = {  class-name@conceptview-name,… };
   fied-name = value;
   …
>,…
}
```

The reference to another instance in an instance description can be achieved through either the complete description of the instance (which is then created), or a reference to the instance in the form:

```
#<<concept keyvalues*>>
```

Comments

Comments (inserted between `/*` and `*/`) can be only put just before an entity description (beginning by a <). These comments are attached to the entities they precede and can be loaded and saved by the system. They can be found under the "`comment`" annotation label.

Backus-Naur form syntax

The following Backus-Naur form is built upon the following conventions:

l-<entity> is a list of <entity>;

entity-name is the name of an <entity>;

keywords are in boldface;

Comments can be expressed with the help of matching /* */

```
<base>          ::=     '<'   base-name
                              required-bases '=' '{' l-file-name '}' ';'
                              types '=' '{' l-desc-type '}' ';'
                              viewpoints '=' '{' l-viewpoint-name '}' ';'
                              concepts '=' '{' l-<concept> '}' ';'
                              methods '=' '{' l-<signature> '}' ';'
                              instance-files '=' '{' l-file-name '}' ';'
                              lexicons '=' '{' l-file-name '}' ';'
                              talk-modules '=' '{' l-file-name '}' ';' '>'
<desc-type>     ::=   typefile-name
<concept>       ::=     '<'   concept-name
                              key '=' '{' l-<conceptslot> '}' ';'
                              fields '=' '{' l-<conceptslot> '}' ';'
                              viewpoints '=' '{' l-<conceptview> '}' ';'
                              bridges '=' '{' l-<bridge> '}' ';' '>'
<concept-slot>::=       '<'   conceptslot-name
                              type '=' <concept> | <type> ';'
                              nature '=' propriete | composant | lien ';'
                              constructor '=' un | liste-de | ensemble-de ';' '>'
<viewpoint>     ::=     '<'   viewpoint-name
                              root-class '=' <root-class> ';'
                              classes '=' '{' l-<class> '}' ';' '>'
<root-class >   ::=     '<'   class-name
                              fields '=' '{' l-<classslot> '}' ';' '>'
<filter>        ::=     <class>
<class>         ::=     '<'   class-name
                              super-class '=' class-name ';'
                              fields '=' '{' l-<classslot> '}' ';' '>'
<classslot>     ::=     '<'   classslot-name <descriptor>* '>'
<descriptor>    ::=           type '=' '{' l-<exp-class> '}' ';'
                  |           domain '=' <domain> ';'
                  |           intervals '=' '{' <interval>+ '}' ';'
                  |           except '=' <domain> ';'
                  |           card '=' <integer> | <interval> ';'
                  |           default '=' <value> ';'
                  |           assign '=' '(' l-<path> ')' ';'
                  |           filters '=' '(' l-<filter> ')' ';'
                  |           methods '=' '(' l-<method> ')' ';'
<bridge>        ::=     '<'   bridge-name
                              source '=' '{' l-<class-ref> '}' ';'
                              destination '=' <class-ref> ';' '>'
<object>        ::=     '<'   is-in '=' '{' l-<class-ref> '}' ';'
                              <field-value>-l '>'
<signature>     ::=   method-name ':' l-<mtype> '->' <mtype>
<mtype>         ::=   <constructeur> <type> | <constructeur> <concept>-name
<constraint>    ::=   constraint-name '(' l-<constraint-term> ')'
<constraint-term>     ::= <argument> | <constraint-operator> '(' l-<constraint-term> ')'
<field-value>   ::=   field-name '=' <value> ';'
<method>        ::=   method-name '(' l-<argument> ')'
<argument>      ::=   <value> | <path>
<path>          ::=   '!' | <path> '.' field-name | <path> ':' field-name
<domain>        ::=   <single-domain> | <collector-domain>
<single-domain> ::= '{' l-<single-value> '}'
<collector-domain> ::= 'P' '(' <collector-value> ')' | '{' l-<collector-value> '}'
<interval>      ::=   <bound> <single-value> <single-value> <boound>
<value>         ::=   <collector-value> | <single-value>
<collector-value>     ::= <set-value> | <list-value>
<list-value>    ::=   '(' l-<single-value> ')'
<set-value>     ::=   '{' l-<single-value> '}'
```

```
<single-value>::=  '|' <anything> '|' | <object-ref>
<class-ref>   ::=  class-name '@' conceptview-name
<object-ref>  ::=  '#<<' concept-name l-<value> '>>'
<boolean>     ::=  true | false
<bound>       ::=  '[' | ']'
```

Keywords and multi-linguism

Multi-linguism is available by setting the `(languages)` TALK form (the first supported language found in this variable is the one considered). Supported languages are currently English and French. Type and constructor keywords are not available in multi-lingual form.

Reserved symbols

[and]	interval delimiters
< and >	structure delimiters
{ and }	set delimiters
(and)	list delimiters
,	set element separator
=	structure item assignment
;	structure item separator
#<< and >>	object reference delimiters
!	path initiator
.	path constructor
:	path constructor with concatenation
->	method functor
==, <>	equality comparators
<, >, >=, <=	ordered types comparators
/* */	comments

Terminals

type	regular expression	meaning
identifiers	[A-Za-z][-_A-Za-z0-9]*	
external value	|[^]*|	anything between | and |
string	\"[^"]*\"	anything between " and "
comments	/*.**/	anything between /* and */
integers	{+-}[0-9]+	
reals	{S}{D}+"."{D}*((e|E){S}{D}+)? {S}"."{D}+((e|E){S}{D}+)?	
boolean	vrai|faux (resp. true|false)	
infinite bounds	[+-]inf	-∞ or +∞

Keywords

The lexical analyser can be used in two exclusive languages: French or English. Here is the exhaustive list of keywords:

base	base
concepts	concepts

attributs	fields
nature	nature
card	card
sauf	except
defaut	default
chemin	assign
methodes	methods
filtres	filters
classes	classes
sorte-de	superclass
classes-depart	sources
est-dans	is-in
points-de-vue	viewpoints
clef	key
dans	type
const	constructor
domaine	domain
intervalles	intervals
condition	condition
classe-racine	root-class
passerelles	bridges
classe-arrivee	destination

Concept field values (only available in French)

propriete	property
composant	component
un	a
liste	list
ensemble	set

API

(**tr-print-when-created** *optional: boolean*) macro in [libtrl]

-> boolean, *boolean* boolean

Sets or returns a value indicating that, while loading a knowledge base, each created entity will cause an indication to be displayed on standard output.

(**tr-load-base** *filename*) function in [libtrl]

-> boolean, *filename* $\text{filename}_{\text{fnf}}$

Loads a knowledge base defined in the file named *filename*. Returns `t` if no error has been detected, `()` otherwise. Results in the knowledge base actually loaded.

(**tr-load-instance-file** *filename*) function in [libtrl]

-> boolean, *filename* $\text{filename}_{\text{fnf}}$

Loads an instance file defined in the file named *filename*. Returns `t` if no error has been detected, `()` otherwise. Results in the instances actually available.

(**tr-unload-instances** *base*) function in [libtrl]

-> boolean, *base* $\text{base}_{\text{base}}$

Deletes all the instances of the concepts of the base. This allows to load again instance files after a failure without having to load again the whole knowledge base.

(**tr-save-base** *base*) function in [libtrl]

-> boolean, *base* base_{nab}

Saves the knowledge base *base* in a file correspondng to its filename. Returns `t` if no error has been detected, `()` otherwise. The initial file, if it exists, is copied with appended "~".

(**tr-save-base-as-talk** *base optional: fileout*) function in [libtrl]

-> boolean, *base* base_{nab}, *filename* $\text{filename}_{\text{fnf}}$

Prints the knowledge base *base* as a TALK program corresponding exactly to the action of loading the knowledge base. The result is printed in the stream *fileout* (by defaults `(stdout)`). Returns `t` if no error has been detected, `()` otherwise.

(**tr-init-xml**) function in [libtrl]

-> boolean

Initialises some variables for XML output. Applying this is required before launching `tr-save-base-as-xml`.

(**tr-save-base-as-xml** *base optional: fileout*) function in [libtrl]

-> boolean, *base* base_{nab}, *filename* $\text{filename}_{\text{fnf}}$

Prints the knowledge base *base* as a XML document corresponding exactly to the contents of the knowledge base. The result is printed in the stream *fileout* (by defaults `(stdout)`). Returns `t` if no error has been detected, `()` otherwise.

(**tr-save-base-as-c** *base optional: fileout*)— function in [libtrl]

-> boolean, *base* base_{nab}, *filename* $\text{filename}_{\text{fnf}}$ NIY

The same as above with a C program instead of a TALK program.

Appendix 3. TROEPS in TROEPS textual format

Here is the description of TROEPS itself in its textual format. This is not the description of the actual implementation. From that description, the instances part have been withdrawn since the instances, being other entities, are implicitly described in the text.

```
/* The knowledge base corresponding to the Troeps kernel */

<base Troeps
  required-bases = {} ;
  types = {"trtdum"};
  viewpoints = {};
  concepts = {
    <concept
      key = {
        <name
          type = chaine; nature = propriete; constructor = un;>};
      fields = {
        <conceptslots
          type = conceptslot; nature = propriete; constructor = ensemble;>,
        <key
          type = conceptslot; nature = propriete; constructor = ensemble;>,
        <viewpoints
          type = conceptview; nature = propriete; constructor = ensemble;>,
        <bridges
          type = bridge; nature = propriete; constructor = ensemble;> };
      viewpoints = {};
      bridges = {};
      >,


    <conceptview
      key = {
        <name
          type = string; nature = propriete; constructor = un;>,
        <concept
          type = concept; nature = propriete; constructor = un;>};
      fields = {
        <root
          type = class; nature = propriete; constructor = un;> ,
        <exhaustive
          type = boolean; nature = propriete; constructor = un;> ,
        <exclusive
          type = boolean; nature = propriete; constructor = un;> ,
        <definitional
          type = boolean; nature = propriete; constructor = un;> };
      viewpoints = {};
      bridges = {};
      >,

  <class
      key = {
        <name
          type = chaine; nature = propriete; constructor = un;>,
        <viewpoint
          type = point-de-vue; nature = propriete; constructor = un;>};
      fields = {
```

```
      <slots
         type = classslots; nature = propriete; constructor = ensemble;>,
      <super-class
         type = class; nature = propriete; constructor = un;>,
      <sub-classes
         type = class; nature = propriete; constructor = ensemble;>,
      <attached
         type = object; nature = propriete; constructor = ensemble;>,
      <constraints
         type = object; nature = propriete; constructor = ensemble;>,
      <bridge-in
         type = bridge; nature = propriete; constructor = ensemble;>,
      <bridge-out
         type = bridge; nature = propriete; constructor = ensemble;> };
   viewpoints = {};
   bridges = {};
   >,


<conceptslot
   key = {
      <name
         type = chaine; nature = propriete; constructor = un;>,
      <concept
         type = concept; nature = propriete; constructor = un;>};
   fields= {
      <viewpoints
         type = viewpoint; nature = propriete; constructor = ensemble;>,
      <type
         type = concept; nature = propriete; constructor = un;>,
      <nature
         type = string; nature = propriete; constructor = un;>,
      <constructor
         type = string; nature = propriete; constructor = un;>
   <method
         type = string; nature = propriete; constructor = ensemble;> };
   viewpoints = {
      <conceptslot-view
         root-class =
            <conceptslot
               fields = {
                  <nature
                     domain = {"propriete", "composant", "lien"}; >,
                  <constructor
                     domain = {"un", "liste", "ensemble"}; >,
                          }>;
         classes = {};
         }};
   bridges = {};
   >,

<classslot
   key = {
      <name
         type = chaine; nature = propriete; constructor = un;>,
      <class
         type = class; nature = propriete; constructor = un;>};
   fields = {
      <type
```

```
            type = class; nature = propriete; const = ensemble;>,
          <domain
            type = value; nature = propriete; const = ensemble;>,
          <intervals
            type = interval; nature = propriete; const = ensemble;>,
          <except
            type = value; nature = propriete; const = ensemble;>,
          <card
            type = interval; nature = propriete; const = un;>,
          <default
            type = value; nature = propriete; const = un;>,
          <assign
            type = path; nature = propriete; const = liste;>,
          <filters
            type = filter; nature = propriete; const = liste;>,
          <methods
            type = method; nature = propriete; const = liste;>,
          <conditions
            type = constraint; nature = propriete; const = liste;>};
        viewpoints = {};
        bridges = {};
        >,

      <bridge
        key = {
          <concept
            type = concept; nature = propriete; constructor = un;>,
          <name
            type = chaine; nature = propriete; constructor = un;>};
        fields = {
          <source
            type = class; nature = propriete; constructor = ensemble;>,
          <destination
            type = class; nature = propriete; constructor = un;>};
        viewpoints = {};
        bridges = {};
        >};
    methods = {} ;
    instance-files = {} ;
    lexicons = {} ;
    talk-modules = {} ;
>
```

Appendix 4. TROEPS DTD

Here is the Document type description for Troeps. It can be found under
http://co4.inrialpes.fr/xml/troeps.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- troeps.dtd version="0.6" last-modification="09/07/1999"
               url="http://co4.inrialpes.fr/xml/troeps.dtd" -->

<!ENTITY % naming "name NMTOKEN #REQUIRED
                   xml:lang NMTOKEN #IMPLIED
                   xmlns:trp CDATA #FIXED 'http://co4.inrialpes.fr/xml/troeps.dtd'">
<!ENTITY % aval "trp:VAL*|trp:OBJREF*">
<!ENTITY % domain "trp:VAL+|trp:OBJREF+|trp:SET+|trp:LIST+">
<!ENTITY % val "trp:VAL|trp:OBJREF|trp:SET|trp:LIST">

<!ELEMENT trp:TRPKB (trp:TRPKBDSC)>
<!ELEMENT trp:TRPKBREF EMPTY>
<!ATTLIST trp:TRPKBREF %naming;>
<!ELEMENT trp:TRPKBDSC (trp:CONCEPTDSC*)>
<!ATTLIST trp:TRPKBDSC %naming;>

<!ELEMENT trp:CONCEPT (trp:TRPKBREF,trp:CONCEPTDSC)>
<!ELEMENT trp:CONCEPTREF EMPTY>
<!ATTLIST trp:CONCEPTREF %naming;>
<!ELEMENT                                                                    trp:CONCEPTDSC
(trp:KEYDSC,trp:KFIELDDSC*,trp:VIEWDSC*,trp:BRIDGEDSC*,trp:OBJDSC*)>
<!ATTLIST trp:CONCEPTDSC %naming;>

<!ELEMENT trp:KEYDSC (trp:KFIELDDSC+)>

<!ELEMENT trp:KFIELD (trp:CONCEPTREF,trp:KFIELDDSC)>
<!ELEMENT trp:KFIELDDSC (trp:CONCEPTREF|trp:TYPEREF)>
<!ATTLIST trp:KFIELDDSC %naming;
                    nature (property|component|link) "property"
                    constructor (un|set|list) "un">
<!ELEMENT trp:KFIELDREF (trp:CONCEPTREF)>
<!ATTLIST trp:KFIELDREF %naming;>

<!ELEMENT trp:VIEW (trp:CONCEPTREF,trp:VIEWDSC)>
<!ELEMENT trp:VIEWDSC (trp:CLASSDSC,trp:CLASSDSC*)>
<!ATTLIST trp:VIEWDSC %naming;>
<!ELEMENT trp:VIEWREF (trp:CONCEPTREF)>
<!ATTLIST trp:VIEWREF %naming;>

<!ELEMENT trp:CLASS (trp:VIEWREF,trp:CLASSDSC)>
<!ELEMENT trp:CLASSDSC (trp:CLASSREF?,trp:CFIELDDSC*)>
<!ATTLIST trp:CLASSDSC %naming;>
<!ELEMENT trp:CLASSREF (trp:VIEWREF)>
<!ATTLIST trp:CLASSREF %naming;>

<!ELEMENT trp:FILTER (trp:FILTERDSC)>
<!ELEMENT trp:FILTERDSC ((trp:CONCEPTREF|trp:CLASSREF*),trp:CFIELDDSC*)>

<!ELEMENT trp:CFIELD (trp:CLASSREF,trp:CFIELDDSC)>
<!ELEMENT                                                                  trp:CFIELDDSC
((trp:TYPEDSCTR|trp:DOMAINDSCTR|trp:EXCEPTDSCTR|trp:INTERVALSDSCTR|trp:CARDINALDSCTR
                |trp:DEFAULTDSCTR|trp:ASSIGNDSCTR|trp:FILTERDSCTR|trp:METHODDSCTR)*)>
<!ATTLIST trp:CFIELDDSC %naming;>
<!ELEMENT trp:CFIELDREF (trp:CLASSREF)>
<!ATTLIST trp:CFIELDREF %naming;>

<!ELEMENT trp:TYPEDSCTR ((trp:FILTER|trp:CLASSREF)+)>
<!ELEMENT trp:DOMAINDSCTR (%domain;)>
<!ELEMENT trp:EXCEPTDSCTR (%domain;)>
<!ELEMENT trp:INTERVALSDSCTR (trp:INTERV+)>
```

```
<!ELEMENT trp:CARDINALDSCTR (trp:INTERV)>
<!ELEMENT trp:DEFAULTDSCTR (%val;)>
<!ELEMENT trp:ASSIGNDSCTR (trp:PATH+)>
<!ELEMENT trp:FILTERDSCTR (trp:FILTER+)>
<!ELEMENT trp:METHODDSCTR (trp:TASK+)>

<!ELEMENT trp:BRIDGE (trp:CONCEPTREF,trp:BRIDGEDSC)>
<!ELEMENT trp:BRIDGEDSC (trp:CLASSREF,trp:CLASSREF+)>
<!ATTLIST trp:BRIDGEDSC %naming;>
<!ELEMENT trp:BRIDGEREF (trp:CONCEPTREF)>
<!ATTLIST trp:BRIDGEREF %naming;>

<!ELEMENT trp:OBJECT (trp:CONCEPTREF,trp:OBJDSC)>
<!ELEMENT trp:OBJDSC (trp:CLASSREF*,trp:ATTVAL+)>
<!ELEMENT trp:OBJREF (trp:CONCEPTREF,trp:ATTVAL+)>

<!ELEMENT trp:ATTVAL (%val;)>
<!ATTLIST trp:ATTVAL %naming;>

<!ELEMENT trp:PATH (trp:CONCEPTREF,trp:PATHLINE*)>
<!ELEMENT trp:PATHLINE (trp:KFIELDREF)>
<!ATTLIST trp:PATHLINE comp (coll|conc) "conc">

<!ELEMENT trp:TASK ((trp:PATH|%val;)*)>
<!ATTLIST trp:TASK %naming;>

<!ELEMENT trp:INF EMPTY>
<!ELEMENT trp:INTERV ((trp:VAL|trp:INF),(trp:VAL|trp:INF))>
<!ATTLIST trp:INTERV begin (open|closed) "closed"
                end (open|closed) "closed">

<!-- Constructors -->
<!ELEMENT trp:SET (%aval;)>
<!ELEMENT trp:LIST (%aval;)>

<!-- External types and values -->
<!ELEMENT trp:TYPEREF EMPTY>
<!ATTLIST trp:TYPEREF %naming;>
<!-- THIS PCDATA CAN BE REPLACED WHEN XML-SCHEMA IS READY?
     NOT SURE FOR THE SAME PROBLEM OF DYNAMICITY... -->
<!ELEMENT trp:VAL (#PCDATA)>
```

Appendix 5. Adding data types to TROEPS

TROEPS comes with a set of primitive data types declared in the `trtdum` module. This module must be loaded in order to uses the corresponding types (`entier`, `chaine`, `reel`, `booleen`).

In TROEPS, primitive data types are declared in a general type system which provides the basic operations of type checking but also those of sub-type relationship inference. This is the basis for building classification mechanisms, type checking and dissimilarity computation. This type system is extensible. So, new types can be added through the definition of an abstract data type (ADT) and will behave just like basic types.

The procedure for using new types is the following:

- Declaring an abstract data type;
- Using this type in a concept field (see the "concept field" section);
- Building up type expressions stored in class fields (see the "class field" section);
- Using values of the types in an instance, classifying the instance, etc.

These steps are further detailed below.

Categories of data types and their properties

Data types are TALK structures. They are presented here for illustration purposes. However, for being recognised by TROEPS they must be declared as below.

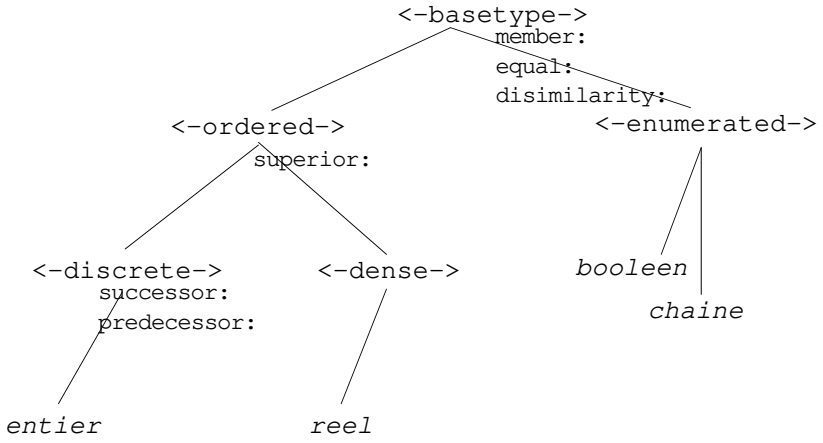


Figure 16: The abstract data type hierarchy in the TROEPS type system. The types in italics are those defined in the `trtdum` standard type module; other types can be added.

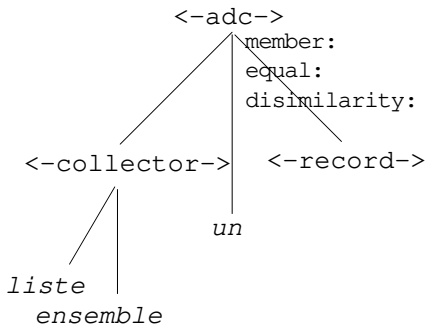


Figure 17: The abstract data constructor hierarchy in the TROEPS type system. The constructors in italics are those defined in the `trtdum` standard type module; it not currently possible to add type constructors to TROEPS.

Declaring data types

An ADT is generally defined through a membership predicate (for deciding if something belongs to the type), an equality predicate (for testing the equality of two values of the type), a string function which returns a string representing a value of the type (for printing purpose). Moreover, a similarity function returns a measure of similarity between two values of the type and ordered types must have an order predicate.

For instance, the date abstract data type can be described with `datep` as membership predicate, `date-equal-p` for equality predicate and `date-anterior-p` as order function. The following table represents the definition of basic data types available in `trtdum`:

name	class	membership	equality	order	successor/ predecessor	dissimilarity	printer
`entier`	`discrete`	`intp`	`i=`	`i>`	`i1+/i1-`	`iabs o i-`	`string`
`reel`	`dense`	`floatp`	`f=`	`f>`	`-`	`fabs o f-`	`string`
`booleen`	`enumerated`	`true`	`eq:t?()`	`-`	`-`	`eq:0.?1.`	`string`
`chaine`	`enumerated`	`stringp`	`string=`	`-`	`-`	`eq:0.?1.`	`id`

TROEPS also uses two collection constructors: list (`liste`) and set (`ensemble`). A collection type is built from a collection operator and a basic type. Collectors are declared through abstract data collectors which enable to check if a collection corresponds to its type, to compare two collections, to order them and, if required, to determine the size of a collection (for cardinality checking).

In the current version of TROEPS, it is not possible to add data collectors.

Using data types

Once data types have been declared, it is possible to describe fields for which the values can be values of the new data type. For that purpose, the type descriptor of the concept field has to be filled by the corresponding ADT (see the "concept" section). Once this has been achieved, it is possible to restrict the domain of the field in particular classes with the help of the `domain`, `intervals`, `except` or `card` descriptors of the class fields (see the "class" section). These type expressions can be expressed in the knowledge representation language as follows:

$$typexp(ADT) = \left\{ \begin{array}{c} \texttt{domain}\ \{v_1,\ldots v_n\} \\ \texttt{except}\ \{v_1,\ldots v_n\} \\ \texttt{intervals}\ \{[v_1\ v_1']\ldots[v_n\ v_n']\} \\ \texttt{card}\ [r\ r'] \end{array} \right\}^*$$

where v are values of the ADT and $r \in N^+ \cup \{+\infty\}$.

Now, instances of the concept can be created: the type system takes care of the membership of the field value to the type; they can be classified, it takes care of the membership of the value to the domain represented by the class field descriptors; new classes can be categorised, the sub-type relationship between their respective field definition is computed by the type module.

Examples

As an example, it is possible to create an abstract data type for the time type provided with TALK. Thanks to the generic arithmetic, it is already possible to test equality (=) superiority (>) and increment (1+) of the time structures. It is also possible to test the membership to the type (timep) and to print time values (string). All this is communicated to TROEPS by only one TALK statement which can be compiled and loaded:

```
(tr-create-adt <-discrete-> time
   member:       timep
   equal:        equal
   superior:     >
   string:       string
   successor:    1+
   predecessor:  1-
   disimilarity: (lambda (v1 v2) (float (abs (- v1 v2))))
   printer:      print
   reader:       read
)
```

API

Services provided by ADT

The only necessary primitive for extending the type system is `tr-create-adt`.

(**tr-create-adt** *adt name rest: listdesc*) macro in [libtrp]

-> adt, *adt* adt, *name* $\text{string}_{\text{nas}}$, *listdesc* [key value]*

Creates a new ADT of kind *adt* and name *name* and returns it. The attributes of the ADT are provided through a set of descriptors (*listdesc*) whose meaning is given by the following:

key	applicability	value	meaning
`member:`	all	anything → B	Does the value belong to the ADT?
`equal:`	all	V×V → B	Are the values equal?
`string:`	all	V → string	The string representation of a value
`disimilarity:`	all	V×V → float	The dissimilarity measure between two values
`superior:`	ordered	V×V → B	Does the first value precede the second one?
`successor:`	discrete	V → V	The successor of a value
`predecessor:`	discrete	V → V	The predecessor of a value
`reader:`	all	strm → V	The reading of a value in a stream
`printer:`	all	V×strm → void	The printing of a value in readable format

These primitives can be exported to anyone.

(**tr-find-adt** *name*) function in [libtrp]

-> adt, *name* $\text{string}_{\text{nas}}$

Retrieves and returns the ADT of name *name*. When no such ADT is known, returns `()`.

(**tr-adt-p** *anything*) function in [libtrp]

-> boolean, *anything* anything

Returns *anything* if *anything* is an ADT, `()` otherwise.

(**tr-adt-top** *name*) function in [libtrp]

-> type, *name* $\text{string}_{\text{nas}}$

Returns the type which corresponds to the domain of all the members of the ADT *adt*.

Services provided for types and values

These primitives are supported for types declared through ADT. The type argument is the one stored into the concept and class fields (e.g. the result of the call to `(.conceptslot-type conceptslot)`). These primitives are used for programming efficiently the classification and categorisation systems. It is possible to use them when only an ADT (and not a type) is available by retrieving the most general type (through `(tr-adt-top adt)`).

(**tr-type-p** *anything*) macro in [libtrp]

-> boolean, *anything* anything

Returns *anything* if *anything* is a type, `()` otherwise.

(**tr-type-member-p** *type value*) function in [libtrp]

-> boolean, *type* type, *value* anything

Checks the membership of value to the type (domain) *type*. Returns *value* if it is type compliant, `()` otherwise.

(**tr-type-equal-p** *type value1 value2*) function in [libtrp]

-> boolean, *type* type, *value1* anything, *value2* anything

Tests the equality of two values declared through the type *type*. Returns *value1* if *value1* and *value2* are equal (in the sense of the ADT equal predicate) and `()` otherwise.

(**tr-type-disimilarity** *type value1 value2*) function in [libtrp]

-> float, *type* type, *value1* anything, *value2* anything

Computes a dissimilarity measure (in the sense of the ADT dissimilarity function) for two values declared through the type *type* and returns it.

(**tr-type-cardinal** *type value*) function in [libtrp]

-> integer, *type* type, *value* anything

Computes a size measure (in the sense of the ADC size function) for a value declared through the type *type* and returns it.

(**tr-type-successor** *type value*) function in [libtrp]

-> anything, *type* type, *value* anything

Computes the successor (in the sense of the ADT successor function) for a value declared through the type *type* and returns it.

(**tr-type-predecessor** *type value*) function in [libtrp]

-> anything, *type* type, *value* anything

Computes the predecessor (in the sense of the ADT predecessor function) for a value declared through the type *type* and returns it.

(**tr-type-superior-p** *type value1 value2*) function in [libtrp]

-> boolean, *type* type, *value1* anything, *value2* anything

Tests the superiority of two values declared through the type *type*. Returns *value1* if *value1* is superior (in the sense of the ADT order predicate) to *value2* and `()` otherwise.

(**tr-type-string** *type value*) generic in [libtrp]

-> string, *type* type, *value* anything

Returns a string corresponding to the representation (in the sense of the ADT string function) of the value *value* declared through the type *type*.

(**tr-type-read-value** *type string*) function in [libtrp]

-> anything, *type* type, *string* string

Returns the value of type *type* corresponding to the string *string*.

(**troeps.type.compare** *type1 type2*) — generic in [libtrp]

-> symbol, *type1* type, *type2* type NIY

Checks the sub-typing of the type (domain) *type1* with regard to that of *type2*. Returns *type1* if *type1* is a super-type of *type2*, *type2* if it is a super-type of *type1*, their intersection otherwise (which can be `()`).

Accessing TROEPS types

TROEPS primitives allows to access the types stored by the type system (in order, for instance, to compare them). These accessors are provided by the representation model in order to store the information related to the type system. They are used in read-write mode by the type module and in read mode by the any other module.

(**troeps.type.class-type** *class*) function in [libtrp]

(**troeps.type.concept-type** *concept*) function in [libtrp]

(**troeps.type.classslot-type** *classslot*) function in [libtrp]

(**troeps.type.conceptslot-type** *conceptslot*) function in [libtrp]

-> type, *class* class,*concept* concept, *classslot* classslot, *conceptslot* conceptslot, *type* type

Returns the type associated to the entity.

Appendix 6. Error system

TROEPS generates error messages which are Ilog TALK conditions. These messages are emitted through the `signal` primitive. Thus, if you want to catch errors in order to manipulate or display them, you must trap these errors or design your own error handler. However, the set of primitives described here is suited for almost every use. Since TROEPS error system is based on TALK objects and generic functions, it is thus easily extendible through refinement.

Note that this interface, unlike the TROEPS API, is made of a number of macro-functions.

Categories of error

An error or warning is a condition (see the TALK reference manual) whose class is a subclass of the `<troeps-error>` class. It thus shares the condition slots:

- the error message associated to the class (`message`);
- arguments to be passed to the error printing function (`message-arguments`);
- the name of the function which raised the error (`caller`);

Other relevant slots of these objects are:

- a possible sub-error (`sub-error`);
- a value for recovery purpose (`continuation-value`).

All these classes are subclasses of the leaves of the hierarchy below.

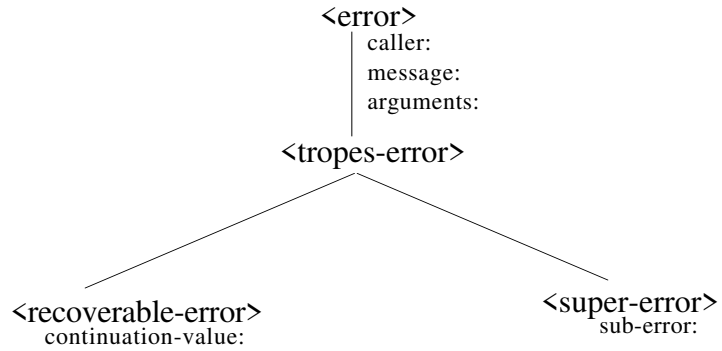


Figure 18: The hierarchy of errors in the TROEPS error system. It can be extended by the user.

Raising errors

An error can be raised by one of the error raising functions. They create the corresponding error instance and signal it. Each error raising function differs from the set of arguments to be provided (a sub-error or a continuation value).

(**troeps.error.raise-error** *type largs funname*) macro in [libtrp]

Exit, *type* symbol, *largs* (sexp*), *funname* symbol

Signals an error of type *type* with the corresponding message arguments *largs* during the call of function *funname*.

(**troeps.error.raise-super-error** *type largs funname suberror*) macro in [libtrp]

Exit, *type* symbol, *largs* (sexp*), *funname* symbol, *suberror* error

Signals a super error of type *type* with the corresponding message arguments *largs* during the call of function *funname*. The error *suberror* is placed in the corresponding slot of the raised error.

(**troeps.error.raise-recup-error** *type largs funname value*) macro in [libtrp]

Exit, *type* symbol, *largs* (sexp*), *funname* symbol, *value* sexpr

Signals a recoverable error of type *type* with the corresponding message arguments *largs* during the call of function *funname*. The value *value* is the continuation value if one wishes to recover from the error.

(**troeps.error.raise-super-recup-error** *type largs funname suberror value*)macro in [libtrp]

Exit, *type* symbol, *largs* (sexp*), *funname* symbol, *suberror* error, *value* sexpr

Signal a super recoverable error of type *type* with the corresponding message arguments *largs* and the calling function *funname*. The error *suberror* is placed in the corresponding slot of the raised error and the value *value* is the continuation value if one wish to recover from the error.

Handling errors

TROEPS provides an error handler which prints the errors and does nothing else (i.e. passes the control to the next handler). For recoverable errors, it prints the error message and returns the continuation value to the program. This handler is activated by calling:

```
? (add-handler (function troeps-handler))
```

This handler can be replaced by another one. The revision system (see below) is based on such a handler.

Trapping errors

An error can be trapped for any kind of treatment. A set of error trapping and processing functions are available in order to process the error instance. These functions usually evaluate some expressions and trap the possibly emitted errors. When no error is raised, its value is returned, when an error is raised a lambda-expression is used for processing the error message.

(**troeps.error.check** *expr type largs funname*) macro in [libtrp]

-> expr, *expr* sexp, *type* symbol, *largs* (sexp*), *funname* symbol

Checks the truth of the evaluation of *expr*. If *expr* is not satisfied, raises an error of type *type* with the corresponding *largs* and *funname*.

(**troeps.error.encapsulate** *expr type largs funname*) macro in [libtrp]

-> expr, *expr* sexp, *type* symbol, *largs* (sexp*), *funname* symbol

Encapsulates the evaluation of *expr*. If an error is raised during its evaluation, it is trapped and introduced as a sub-error in a new error of type *type* with corresponding *largs* and *funname*. This new error is raised in turn. If no error is raised during the evaluation of *expr*, the result of this evaluation is returned.

(**troeps.error.trap-error** *expr treatment*) macro in [libtrp]

-> expr, *body* expr, treatment function

Evaluates *expr* and returns the result of the evaluation. If an error is raised during this evaluation, it is trapped and passed as the only argument of the *treatment* function which is processed in turn.

(**troeps.error.immediate** *expr*) macro in [libtrp]

-> expr, *body* expr, treatment function

Evaluates *expr* and returns the result of the evaluation. If an error is raised during this evaluation, this error is trapped and passed as the only argument of the **tr-immediate-error-hook** method which is processed in turn. Note that no more error is raised afterwards. Thus, if an error has to be propagated, this must occur in the body of the **tr-immediate-error-hook** implementation.

(**tr-immediate-error-hook** *error*) generic in [libtrp]

(**tr-immediate-error-hook** *error*) method in [libtrp]

-> expr, *error* error

Proceeds to a **print-error-stack** of the *error*.

Printing errors

An error can be printed with the help of the `print-error` generic function. Any super-error printed by `print-error` only displays the outer error message. However, it is possible to print all the embedded error messages with the help of the `print-error-stack` function.

(**troeps.error.print-error** *error*) generic in [libtrp]

(**troeps.error.print-error** *error*) method in [libtrp]

->, *error* error

Displays the error message corresponding to the *error* on the standard error file (`stderr`).

(**troeps.error.print-error-stack** *error*) generic in [libtrp]

(**troeps.error.print-error-stack** *error*) method in [libtrp]

->, *error* error

Same as **print-error**.

(**troeps.error.print-error-stack** *error*) method in [libtrp]

->, *error* super-error

Recursively displays the stack of all errors in the sub-error slot of the *error* object on the standard error file (`stderr`).

Correcting errors and revising assertions

When some particular errors are triggered Troeps is able to help the user in correcting these errors. The revision mechanism can guide the user through the possible revision of the knowledge base (or the initial query) in order to modify it in a consistent way. The revision system is available through the graphic interface when the variable `*revision-p*` is evaluated to true.

Interface

Screendump 55: Simple error page.

Screendump 56: Non recognised object (or bad URL).

NIY

Screendump 57: Revision page when an inconsistency occurs.

OK:..... see Screendump 58, p. 145

NIY

Screendump 58: Further revision page.

<>:..... see Screendump 58

Appendix 7. Error messages

Here is the list of error messages. The R signifies that the error is recoverable and the S that it is a super error.

```
ada      already declared annotation label: %s
adb      already defined base : %s
adc      already defined class : %s
adcs     already defined class slot : %s
adf      already defined slot descriptor : %s
adk      already defined concept : %s
adks     already defined concept slot : %s
adkv     already defined concept view : %s
adl      already defined lexitem : %s
adm      already declared method: %A
adp      %s: already defined bridge in concept %A
aeo      already existing object for concept %A : %A
avs      slot %A already valued for object %s (%A)
bak      bridge across concepts
bcf      bad weighting coeficient (float between 0. and 1. expected) %A
btm      badly typed method %A
btv   S  badly typed value for slot %A
c-it     incompatible types: %A and %A
caf      cannot attach to filter %A
cbr      the group of bridges %A is inconsistant
ccv   S  %A: class slot constraint violation in class %A for value %A
cdnl     can't deal with nested lists for path %A
cdrc     cannot delete root class %A
cmk      cannot modify key slot
crrc     cannot rename root class %A
esk      cannot create slotless concept %A
fnf      file not found: %A
iac      Non compatible attachment classes: %A
ibr   S  the bridge %A introduces circularities
ick      key %A is not complete for concept %A, received %A
idb   S  ill-defined bridge (unrecoverable error) for %A: %A
ifpk     %A: ill-formed path from %A (no concept %A)
ifpm     %A: ill-formed path from %A (incorrect mode)
ifps     %A: ill-formed path from %A (no conceptslot)
isc      Illegal stream content: %A missing while reading %A
ivat     invalid value %A according to type %A
ivb      %A: invalid interval bound
ivbp     invalid bridge specification: %A
ivc      %A : invalid constructor for conceptslot %A
ivcd     %A : invalid cardinal for classslot %A
ivd      %A : invalid domain for classslot %A
ivdv     %A : invalid default value for classslot %A
ive      %A : invalid except for classslot %A
ivi      %A : invalid interval for classslot %A
ivk   S  invalid key %A for concept %A
ivks     invalid key specification %A
ivm      %A : invalid method for classslot %A
ivn      %A : invalid nature for conceptslot %A
ivp      %A : invalid unary constraint for classslot %A
ivs      %A : invalid split for classslot %A
ivsn  S  invalid slot values for %A, class %A
ivsp     invalid standpoint (list of views): %A
ivsv  S  invalid value for slot %A : %A (object %A)
ivt      %A : invalid type for slot %A
kcv   S  constraint violation for concept slot %A
mck      key %A for concept %A is missing in %A
naa      %A is not an annotation label
nab      %A is not a base
nac      %A is not a class
nacs     %A is not a class slot
nact     %A is not a constraint expression
```

```
naeo     %A is not an existing object
naev     %A is not an 'ensemble' value
naf      %A is not a filename
naft     %A is not a filter
naitv    %A is not an interval
nak      %A is not a concept
naks     %A is not a concept slot
nakv     %A is not a concept view
nal      %A is not a list
nali     %A is not a lexitem
nalv     %A is not a 'liste' value
nao      %A is not an object
naot     %A is not an ordered type
nap      %A is not a bridge
naph     %A is not a path
nas      %A is not a string
nasb     %A is not a symbol
nast     %A is not a stream
nat      %A is not a type
nauv     %A is not an 'un' value
ncs      non constructed slot %A
nct      non constructed type
neo      %A cannot be destroyed (not empty)
nib      %A not in base %A
nic      %A not in classes %A
nics     %A not in class slots %A
nik      %A not in concept %A
niks     %A not in concept slots %A
nikv     %A not in concept views %A
nip      %A not in bridges %A
ntr      string %A not totally read for type %A (rest %A)
perr     Error while parsing file %A line #%i.
rba   R  new bridge %A subsumes bridge %A
rbb   R  new bridge %A is subsumed by bridge %A
scv      class %A is in the same concept-view as another in the list
stv      %A: restriction not compatible with class %A
ucs      constructed slot %A non initialised in %A
uic      non instanciable constructor: %A
uif      non instanciable slotdescriptor: %A
uit      non instanciable type: %A
ukks     unknown concept slot %A for concept %A
w-eqt R  both types are equal
w-ivp R  ':' instead of '.' at position %A in %A
w-str R  strange refinement for slot %A in class %A
w-tld R  prohibited values or intervals
w-tlt R  new type included in the super-type
w-tnj R  type overlapping
w-vat R  value added twice to the set
wcfp     path %A has for constructor %A instead of %A
wtfp     path %A has for type %A instead of %A
```

Appendix 8. Reserved annotation labels

Among the annotation names that can be used in TROEPS, some of them are already reserved names within the system. They are described below. The S column displays the status of the annotations stored here: (r)eserved to the system, (a)utomatically filled by the system (but can be changed by the users) or fully (p)ublic.

label	S	destination
prehtml	p	Contains the filename of an introductory HTML description of the entity (see p. 115)
posthtml	p	Contains the filename of an concluding HTML description of the entity (see p. 115)
funhtml	p	Contains the Talk function used for generating the HTML pages of the objects (see p. 116)
namehtml	r	Contains the cached name of objects for HTML display
lexitem	a	Contains the lexitem associated to an entity
comment	a	Contains the comments read in the base description file
creator	a	Contains the name of the CO_4 creator
date	a	Contains the date of the CO_4 creation
clalgo	r	Contains the algorithm for clustering (see p. 106)
clweight	r	Contains the field weight for clustering (see p. 106)
clview	r	Contains the views to use for clustering on a field (see p. 106)
clpred	r	Clustering reserved
clcirc	r	Clustering reserved
clvisited	r	Clustering reserved
clprocessed	r	Clustering reserved
toexplore	r	Troeps reserved
visited	r	Troeps reserved
argumentation-tree	r	Argumentation reserved: Associated argumentation tree
prec	r	Revision reserved
prec-init	r	Revision reserved
suiv	r	Revision reserved
suiv-init	r	Revision reserved
depl-ref	r	Revision reserved
ref-intouchable	r	Revision reserved
cl-pb	r	Revision reserved
cause-suppress	r	Revision reserved
sl-pb	r	Revision reserved

Appendix 9. Glossary

Abstract data type

An abstract data type specifies the operations which are applicable to its values (and its type expressions). This enables TROEPS to manipulate values which are not objects in a safe way (i.e. that it is able to type and sub-type, to print and to compare them).

ADT

See Abstract data type.

Annotation

Any piece of data attached to a field value or an entity which is not part of the knowledge model but used by super layers of the TROEPS system. A.k.a. meta-values (because they are values about the field value). It can be a comment about the source, confidence or validity range of the value. It can also be service information such as hypertext link or graphic display dependence.

API

Application Programming Interface. A set of primitives which allow to manipulate the implementation of the TROEPS model.

Atomicity

The API is designed in order to implement atomic actions, i.e. actions which turn a valid knowledge base into another valid knowledge base when succeed and leave the initial knowledge base unaltered when fail.

Attached

Said of an object by reference to the class to which it is attached under a particular viewpoint. This class is the more specialised class to which the object belongs (is member of).

Bridge

A bridge is a constraint on the attachement classes of the objects. It constrains the objects member of a particular set of classes to be member of a particular class of another viewpoint.

Bridge inference

The bridge inference mechanisms allow to find all the maximal bridges that can exists between a set of viewpoints and another viewpoint. These inference can be based either on the type of the classes to compare or the set of objects which are their members.

Categorisation (class classification)

Finds the set of classes which can have a particular class as sub-class or super-class.

CGI (script)

CGI (Common Gateway Interface) is a technique for generating dynamic HTML pages in a HTTP server. A program called CGI script is launched by the HTTP server when particular URLs are requested and its output is a HTML page which is redirected on the server output.

Class

A class is a conceptual entity which group instances of a concept sharing some characteristics under a particular viewpoint. It specifies the characteristics to be shared by constraints on the fields of these objects. It is made up of a name and of a set of fields. Each field has a name and a set of descriptors. These descriptors associate to a field its type and, more generally, constraints on the field values for the members of the class. Classes are organised into hierarchies through the specialisation relation.

Classification (instance)

Finds the set of classes to which an object can be attached (i.e. for which the object does not violate some of their constraints). Classification is used to select a method for an elementary task and to characterise a task in order to select a decomposition which is adapted to the context, i.e. the input data of the task.

Classification (taxonomy)

Hierarchy of classes along the specialisation relation. See taxonomy.

Clustering

Clustering is an inference mechanism able to build an entire taxonomy of classes (including the constraints on classes) from the set of objects in the concept.

Co4

Co4 is a system and protocol allowing several users to modify concurrently a knowledge base. It modifies the knowledge base only when all its subscribers have decided to do so.

Component

Object which is also the value of a field of another object whose nature is component.

Composite (object)

Object with at least one field whose nature is component.

Concept

Definition of the ontological aspect of a family of objects. More precisely, the concept deals with the structure and the identity of objects. A concept is described by concept fields descriptors and viewpoints. It also determines the key used in order to identify the objects.

Constructor

In TROEPS, each value is subject to a constructor which allows to gather several simple values into a maybe more complex one. In TROEPS, these constructors can be "un" (which returns the simple data), set and list.

CSP

CSP stands for constraints satisfaction problem. Such a problem consists in finding the possible values in the domain of a set of variables satisfying a set of constraints.

Constraint

Constraint has two meaning in TROEPS. The usual meaning denotes the fact that some constraints apply to objects which are members of classes. The specialised meaning refers to the CSP framework. It allows to set constraints on concepts, classes and objects and to rely on a constraint solver in order to restrict the possible values of the fields on which the constraints applies.

Decomposition

A complex task must be recursively decomposed into sub-tasks, down to elementary tasks. A task is described by a class, the fields of which are associated with its input and output parameters. Its decomposition is described by an operator (sequence, selection, etc.) and the list of tasks on which this operator applies. The decomposition of a task into sub-tasks depends on the type of its input data. Thus, the decomposition step is preceded by a classification step, during which the task is first characterised according to its known input values.

Default value

A default value is an inference mechanism which provides a (constant) hypothetical value for a field to the members of a classes.

Document type description (DTD)

Definition of a set of tags or element by specifying their content (a regular expression based on the other tags which specify a sub-tree), their attributes.

Exclusive

A set of classes is exclusive if any object belonging to one of them does not belong to any other of them. TROEPS is subjectively exclusive.

Exhaustive

A set of classes exhaust their common super-class (which is thus exhaustive) if every object attached to the super-class belongs to one of its sub-classes. TROEPS does not require exhaustiveness.

eXtensible Markup Language

A successor of both HTML and SGML which is a general markup language (without semantics). Its first use is in exchanging structured data. The syntax of particular documents can be specified in a DTD.

Field

A field (a.k.a. slot, feature, attribute) represents an attribute of an object.

Filter

A filter is a shape of class which is used in order to screen the objects, instance of particular classes, which satisfies particular constraints. It can be used as an inference mechanism in order to obtain the value of a field.

HTML

Hyper-text mark-up language. A format for encoding hypertext documents so that they are visible and accessible through a network. HTML is based on the SGML (standard generalised mark-up language) syntax.

HTTP

Hyper-text transmission protocol. The protocol used for dealing with HTML documents through TCP/IP networks.

Hypothetical value

A hypothetical value is a value for a field, returned by an inference mechanism, which is by no mean the real value of the field but some plausible value which has been computed from the inference mechanisms described in classes.

Implicit value

An implicit value is a value for a field, returned by an inference mechanism, which has the same status as the values that have been explicitly put in the object description. The value is computed by an inference mechanism described in concept.

Inference

An inference is a computation which can either provide the value for a field or the attachment classes for an object. It can correspond to an implicit computation (providing the true value) or a hypothetical computation (trying to guess what the true value could be).

Instance

Said of an object by reference to its concept (see also attached).

Instanciation

The mechanism which creates an object. It includes the indexing of the object in its concept under its key and the type checking of field values with regard to the type of all the classes to which it belongs.

IP

Stands for Internet protocol.

Key

Set of fields which identifies uniquely the instances of a concept.

Knowledge base

A knowledge base is a set of concepts and instances of these concepts. It is stored in a file and can import TALK modules, lexicons, other knowledge bases and modules describing types. It can be saved in a textual format, TALK or HTML.

Knowledge server

A knowledge server is an application which distributes the knowledge contained in one or more knowledge bases. TROEPS uses the World-wide web in order to distribute the knowledge across a network.

Label

A tag under which an annotation can be stored and retrieved.

Lexicon

A lexicon is a set of definitions of terms. Each definition corresponds to a particular term and is described in a lexitem. A lexitem provides a definition for the term together with links to other lexitems and TROEPS object.

Lexitem

A lexicon item, that is a computer structure representing a term.

Meta-value

See Annotation.

Method (1)

Knowledge expressing how to obtain new knowledge from the available knowledge. Basically, a method is the way to infer the value of a field. It makes a link between the object aspect of the knowledge base and its task aspect.

Method (2)

An external module implementing an executable task. A method is externally described by a class, the fields of which are associated with the input and output parameters of the module. To process a method, an instance of the class is created and the fields associated with the input arguments are filled in. This instance is then transmitted to the module, the execution of which allows to fill in the output fields. Since methods are modelled as classes, they can be organised into hierarchies in which general methods dominate more specific ones. A method is more specific (resp. more general) than another one if the types of its input arguments are subtypes (resp. super-types) of the types of the arguments of the other one.

Membership (of an object to a class)

An object is a member of a class when it is attached to that class or to one of its sub-classes. It must satisfy all the constraints of the class and can benefit from all inference mechanisms attached to it.

Nature

The nature of a field gives it a particular semantics (and is thus tied to a particular implementation).

Object

The beginning of the whole story. An object is made up of a name and of the list of the known values of the fields of the concept to which it is instance. It can be seen under the concept viewpoints as belonging to some classes. Missing values can be determined using methods declared in the concept field and guessed using the knowledge attached to the class field: functions or default values.

Objective property

A property whose satisfaction can be checked by the system. It can, in particular, reject any operation which would result in a knowledge base not satisfying this property. An example of such a property is the impossibility to create a class which cannot be attached any object.

Path

A path is an expression for describing how to access a value from an object (or from the instances of a concept or the members of a class) through a sequence of field values.

Problem solving process

The process through which a task is solved by either calling a method, or by decomposing recursively the task into sub-tasks.

Revision

The procedure which suggests corrections when an error is raised.

Root class

A class which is the root of a taxonomy. Any instance of a concept is member of the root class in each of its viewpoints. There is a two-way bridge between any two root classes of any viewpoint of the same concept.

Server

See Knowledge server.

Slot

See Field.

Specialisation

Relation from a class to another one of the same viewpoint of a concept which indicates that the objects members to the former are also members to the latter. To that extent, the constraints

attached to the former can only be more constraining than those of the latter.

Subjective property

A property whose satisfaction must be considered by the system, but that the system cannot check. However, it is sometime possible that the system can detect a violation of the property. An example of a subjective property is the assertion that procedural attachments do not have side-effects.

Talk

Programming language of the Lisp family developed by the company Ilog. It translates modules into C, so can be compiled into libraries. The libraries are free of charge as far as they do not contain elements of the programming environment.

Task

The task represents procedural knowledge. It is characterised by a goal to be achieved and a set of knowledge that can be used in order to achieve it. It can be implemented by either a procedure to be called or a set of sub-tasks which constitutes the concurrent or dependant ways to achieve it. A task is associated with a problem. To solve the task is then to solve the problem. An elementary task is solved by the execution of a method. The selection of the adapted method among the set of available methods which is attached to the elementary task results from a classification process. A terminal task is executed as a method while a non-terminal one is decomposed.

Taxonomy

A taxonomy is a partially ordered set of entities (here the classes ordered by specialisation) which are interpreted as sets of other entities (here the objects attached to each class) such that the partial order is compatible with the inclusion of interpretations.

Troeps

Has no particular meaning.

Tropes

The previous name of the TROEPS system. Changed for juridical reason.

Type

See Type expression and Abstract data type.

Type expression

The domain of value of a particular role to be played by an entity of the system. The type is defined with the help of descriptors for field values and by the definitions of concepts and classes.

URL

A URL (Uniform Resource Locator) is the address by which the documents are identified in the World-wide web.

Value passing

Value passing is an inference mechanism which provides a value for a field to the members of a class through the specification of a path which allows to access the value.

Viewpoint

Conceptual position from which only a projection of the knowledge base is available. This position enables to access only to certain fields of the objects of a concept. It also organises classes in concurrent hierarchies. These hierarchies differ in the description of the classes they incorporate and/or in the way classes and sub-classes are related. An object may belong to one class on each viewpoint. It is thus possible to see an object from a single viewpoint or from several viewpoints simultaneously. Different fields will then be available.

Visibility

The fact that a field of an object can be seen (queried, modified, etc.) under a particular viewpoint.

WWW

Acronym of World-wide web.

XML

Acronym of eXtensible Markup Language.

Appendix 10. Function index

By using the on-line version of the manual, you should be able to retrieve the function definitions through the “Find” facility of your HTML browser.

(nrk-reply keywork socket) function in [libtrh]....118
(tr-add-classslot-assign classslot paths) function in [libtrp] 90
(tr-add-classslot-filters classslot filter) function in [libtrp].... 91
(tr-add-classslot-method classslot method) function in [libtrp] 92
(tr-add-concept base concept) function in [libtrp].... 16
(tr-add-conceptslot-method conceptslot method) function in [libtrp].... 92
(tr-add-import base base2) function in [libtrp] 16
(tr-add-instance-file base filename) function in [libtrp] 16
(tr-add-lexicon base filename) function in [libtrp].... 17
(tr-add-module base modulename) function in [libtrp] 17
(tr-add-type base type) function in [libtrp] 16
(tr-add-value object slotname value rest: views) function in [libtrp].... 71
(tr-add-viewpoint base viewpointname) function in [libtrp].... 15
(tr-adt-p anything) function in [libtrp] 138
(tr-adt-top name) function in [libtrp] 138
(tr-attach object class) function in [libtrp] 70
(tr-attachable object class) function in [libtrp].... 98
(tr-attach-lexicon base) function in [libtrp] 81
(tr-base basename) macro in [libtrp].... 15
(tr-base-concepts base) function in [libtrp].... 16
(tr-base-filename base) function in [libtrp].... 15
(tr-base-imports base) function in [libtrp].... 16
(tr-base-instance-files base) function in [libtrp].... 17
(tr-base-lexicons base) function in [libtrp].... 17
(tr-base-modules base) function in [libtrp].... 17
(tr-base-name base) function in [libtrp].... 15
(tr-base-p anything) predicate in [libtrp].... 15
(tr-base-rename base string) function in [libtrp].... 15
(tr-base-sorted-concepts base optional: sort-function) function in [libtrp].... 16
(tr-base-types base) function in [libtrp].... 16
(tr-base-viewpoints base) function in [libtrp] 16
(tr-bridge concept bridgename source destination) function in [libtrp].... 63
(tr-bridge-concept bridge) function in [libtrp] 63
(tr-bridge-destination bridge) function in [libtrp].... 63
(tr-bridge-name bridge) function in [libtrp] 63
(tr-bridge-p anything) predicate in [libtrp].... 62
(tr-bridge-rename bridge string) function in [libtrp] 63
(tr-class conceptview classname superclass) macro in [libtrp].... 42
(tr-class-add-constraint class constraint)— function in [libtrp] 95
(tr-class-all-subclasses class) function in [libtrp].... 43
(tr-class-attached class) function in [libtrp] 44

(tr-class-bridges-in class) function in [libtrp]..... 43
(tr-class-bridges-out class) function in [libtrp]..... 43
(tr-class-concept class) function in [libtrp]..... 43
(tr-class-conceptview class) function in [libtrp] 43
(tr-class-constraints class)— function in [libtrp]..... 95
(tr-class-filters class) function in [libtrp]..... 57
(tr-classify object rest: views) function in [libtrp]..... 98
(tr-class-members class) function in [libtrp]..... 44
(tr-class-name class) function in [libtrp]..... 43
(tr-class-p anything) predicate in [libtrp] 42
(tr-class-proper-filters class) function in [libtrp]..... 57
(tr-class-rem-constraint class constraint-exp)— function in [libtrp]..... 95
(tr-class-rename class string) function in [libtrp]..... 43
(tr-classslot class classslotname) macro in [libtrp] 51
(tr-classslot-class classslot) function in [libtrp] 52
(tr-classslot-conceptslot classslot) function in [libtrp]..... 52
(tr-classslot-name classslot) function in [libtrp]..... 52
(tr-classslot-p anything) predicate in [libtrp] 51
(tr-class-slots class) function in [libtrp] 43
(tr-class-sorted-slots class optional: sort-function) function in [libtrp]..... 43
(tr-class-subclasses class) function in [libtrp]..... 43
(tr-class-superclass class) function in [libtrp] 43
(tr-class-superclass-p superclass class) function in [libtrp] 43
(tr-clustering concept method create-p viewname slots) function in [libtrt]..... 107
(tr-concept conceptname key) macro in [libtrp]..... 25
(tr-concept-add-constraint concept constraint)— function in [libtrp]..... 95
(tr-concept-bridges concept) function in [libtrp]..... 26
(tr-concept-constraints concept)— function in [libtrp]..... 95
(tr-concept-instances concept) function in [libtrp] 26
(tr-concept-key concept) function in [libtrp]..... 25
(tr-concept-name concept) function in [libtrp] 25
(tr-concept-p anything) predicate in [libtrp]..... 24
(tr-concept-proper-filters concept) function in [libtrp] 57
(tr-concept-rem-constraint concept constraint)— function in [libtrp]..... 95
(tr-concept-rename concept string) function in [libtrp] 25
(tr-conceptslot concept conceptslotname type const nature) macro in [libtrp]..... 31
(tr-conceptslot-concept conceptslot) function in [libtrp] 31
(tr-conceptslot-name conceptslot) function in [libtrp] 31
(tr-conceptslot-p anything) predicate in [libtrp]..... 30
(tr-conceptslot-rename conceptslot string) function in [libtrp] 31
(tr-concept-slots concept) function in [libtrp]..... 25
(tr-conceptslot-viewpoints conceptslot) function in [libtrp]..... 31
(tr-concept-sorted-instances concept optional: sort-function) function in [libtrp]..... 26
(tr-concept-sorted-slots concept optional: sort-function) function in [libtrp] 25
(tr-concept-sorted-views concept optional: sort-function) function in [libtrp]..... 25
(tr-conceptview concept conceptviewname) macro in [libtrp]..... 36

(tr-conceptview-concept conceptview) function in [libtrp]..... 36
(tr-conceptview-definitional conceptview optional: boolean) function in [libtrp] 37
(tr-conceptview-exclusive conceptview optional: boolean) function in [libtrp]..... 36
(tr-conceptview-exhaustive conceptview optional: boolean) function in [libtrp] 36
(tr-conceptview-name conceptview) function in [libtrp]..... 36
(tr-conceptview-p anything) predicate in [libtrp] 35
(tr-conceptview-rename conceptview string) function in [libtrp]..... 36
(tr-conceptview-root conceptview) function in [libtrp] 36
(tr-concept-views concept) function in [libtrp]..... 25
(tr-create-adt adt name rest: listdesc) macro in [libtrp] 138
(tr-create-base basename optional: filename) function in [libtrp] 15
(tr-create-bridge concept name source destination) function in [libtrp] 62
(tr-create-class superclass classname) function in [libtrp]..... 42
(tr-create-classslot class classslotname) function in [libtrp]..... 51
(tr-create-complete-object concept classlist slotvallist) function in [libtrp]..... 69
(tr-create-concept conceptname key) function in [libtrp] 24
(tr-create-conceptslot concept conceptslotname type const nature) function in [libtrp] 30
(tr-create-conceptview concept conceptviewname) function in [libtrp] 36
(tr-create-filter concept rest: classes) function in [libtrp]..... 57
(tr-create-lexitem name hyperonyms) function in [libtrp]..... 79
(tr-create-object concept keyval) function in [libtrp] 69
(tr-current-base) setfable function in [libtrp]..... 17
(tr-declare-annotation label comment optional
type const) function in [libtrp]..... 74
(tr-declared-annotation-p label) function in [libtrp] 74
(tr-declare-method name args result) function in [libtrp] 92
(tr-delete-base base) function in [libtrp]..... 15
(tr-delete-bridge bridge) function in [libtrp]..... 63
(tr-delete-class class) function in [libtrp] 42
(tr-delete-classslot classslot) function in [libtrp] 51
(tr-delete-concept concept) function in [libtrp]..... 25
(tr-delete-conceptslot conceptslot) function in [libtrp]..... 31
(tr-delete-conceptview conceptview) function in [libtrp] 36
(tr-delete-filter filter) function in [libtrp] 57
(tr-delete-lexitem lexitem) function in [libtrp]..... 79
(tr-delete-object object) function in [libtrp]..... 69
(tr-erase-base base) function in [libtrp]..... 15
(tr-erase-bridge bridge) function in [libtrp]..... 63
(tr-erase-class class) function in [libtrp] 42
(tr-erase-classslot classslot) function in [libtrp] 52
(tr-erase-concept concept) function in [libtrp]..... 25
(tr-erase-conceptslot conceptslot) function in [libtrp]..... 31
(tr-erase-conceptview conceptview) function in [libtrp] 36
(tr-erase-filter filter) function in [libtrp] 57
(tr-erase-object object) function in [libtrp]..... 70
(tr-filtered-objects filter) function in [libtrp]..... 57

(tr-filter-p anything) function in [libtrp]..... 56
(tr-find-adt name) function in [libtrp].....138
(tr-find-base basename) function in [libtrp]..... 15
(tr-find-bridge concept bridgename) function in [libtrp] 62
(tr-find-bridges-from-objects classes class created-p) function in [libtrt] 108
(tr-find-bridges-from-types classes class created-p) function in [libtrt].....109
(tr-find-class conceptview classname) function in [libtrp] 42
(tr-find-classslot class classslotname) function in [libtrp] 51
(tr-find-concept conceptname) function in [libtrp] 24
(tr-find-conceptslot concept conceptslotname) function in [libtrp] 31
(tr-find-conceptview concept conceptviewname) function in [libtrp]..... 36
(tr-find-lexitem name) function in [libtrp] 79
(tr-find-object concept keyval) function in [libtrp]..... 69
(tr-find-slot-similar object slots) function in [libtrt] 107
(tr-find-view-similar object views) function in [libtrt].....107
(tr-get-annotation entity label) function in [libtrp] 74
(tr-get-classslot-assigns classslot) function in [libtrp]..... 90
(tr-get-classslot-card classslot) function in [libtrp]..... 53
(tr-get-classslot-condition classslot)— function in [libtrp]..... 96
(tr-get-classslot-defaults classslot) function in [libtrp]..... 88
(tr-get-classslot-domain classslot) function in [libtrp]..... 52
(tr-get-classslot-except classslot) function in [libtrp] 52
(tr-get-classslot-filters classslot) function in [libtrp]..... 91
(tr-get-classslot-intervals classslot) function in [libtrp] 52
(tr-get-classslot-methods classslot) function in [libtrp]..... 92
(tr-get-classslot-type classslot) function in [libtrp]..... 52
(tr-get-conceptslot-constructor conceptslot) function in [libtrp]..... 31
(tr-get-conceptslot-methods conceptslot) function in [libtrp] 92
(tr-get-conceptslot-nature conceptslot) function in [libtrp] 32
(tr-get-conceptslot-type conceptslot) function in [libtrp]..... 31
(tr-get-lexitem entity label) function in [libtrp] 81
(tr-get-slot-annotation object slotname label) function in [libtrp]..... 74
(tr-get-value object slotname rest: views) function in [libtrp]..... 70
(tr-guess-value object slotname rest: views) — function in [libtrp] 88
(tr-immediate-error-hook error) generic in [libtrp].....143
(tr-immediate-error-hook error) method in [libtrp] 143
(tr-init) function in [libtrp]..... 10
(tr-init-xml) function in [libtrl].....130
(tr-is-a object) macro in [libtrp]..... 70
(tr-is-in object rest: views) function in [libtrp]..... 70
(tr-is-in-under object conceptview) function in [libtrp]..... 70
(tr-lexitem-add-antonym lexitem antonym) function in [libtrp].....80,81
(tr-lexitem-add-bibliography lexitem bibliography) function in [libtrp] 80,81
(tr-lexitem-add-hyponym lexitem hyponym) function in [libtrp].....80,81
(tr-lexitem-add-object lexitem object) function in [libtrp] 80
(tr-lexitem-add-synonym lexitem synonym) function in [libtrp].....80,81

(tr-lexitem-antonyms lexitem) function in [libtrp] 80
(tr-lexitem-bibliography lexitem) function in [libtrp] 80
(tr-lexitem-definition lexitem) function in [libtrp]..... 79
(tr-lexitem-hyperonym lexitem) function in [libtrp] 80
(tr-lexitem-hyponyms lexitem) function in [libtrp] 80
(tr-lexitem-name lexitem) function in [libtrp] 79
(tr-lexitem-object lexitem) function in [libtrp] 80
(tr-lexitem-remove-object lexitem object) function in [libtrp]..... 81
(tr-lexitem-rename lexitem name) function in [libtrp]..... 79
(tr-lexitem-set-definition lexitem definition) function in [libtrp] 80
(tr-lexitem-set-hyperonym lexitem hyperonym) function in [libtrp]..... 80
(tr-lexitem-synonyms lexitem) function in [libtrp]..... 80
(tr-load-base filename) function in [libtrl]..... 129
(tr-load-instance-file filename) function in [libtrl] 129
(tr-load-lexicon filename optional: print) function in [libtrp]..... 79
(tr-make-constraint constraint-predicate rest: constrained-terms)— function in [libtrp]..... 95
(tr-make-constraint-term constraint-operator rest: constrained-terms)— function in [libtrp]..... 95
(tr-make-path conceptslots) function in [libtrp] 90
(tr-mem-value object slotname value rest: views) function in [libtrp] 71
(tr-modify-value object slotname value rest: views) function in [libtrp]..... 71
(tr-object concept keyval) macro in [libtrp]..... 69
(tr-object-add-constraint object constraint)— function in [libtrp]..... 95
(tr-object-concept object) function in [libtrp] 70
(tr-object-constraints object)— function in [libtrp]..... 96
(tr-object-delete-constraint object constraint)— function in [libtrp]..... 96
(tr-object-key object) function in [libtrp]..... 70
(tr-object-p anything) predicate in [libtrp]..... 69
(troeps.error.check expr type largs funname) macro in [libtrp]..... 142
(troeps.error.encapsulate expr type largs funname) macro in [libtrp] 142
(troeps.error.immediate expr) macro in [libtrp]..... 142
(troeps.error.print-error error) generic in [libtrp]..... 143
(troeps.error.print-error error) method in [libtrp] 143
(troeps.error.print-error-stack error) generic in [libtrp]..... 143
(troeps.error.print-error-stack error) method in [libtrp]..... 143
(troeps.error.raise-error type largs funname) macro in [libtrp]..... 141
(troeps.error.raise-recup-error type largs funname value) macro in [libtrp]..... 142
(troeps.error.raise-super-error type largs funname suberror) macro in [libtrp] 141
(troeps.error.raise-super-recup-error type largs funname suberror value) macro in [libtrp] 142
(troeps.error.trap-error expr treatment) macro in [libtrp] 142
(troeps.http.file-object url) function in [libtrh]..... 117
(troeps.http.urldecode url begin end) function in [libtrh]..... 118
(troeps.type.classslot-type classslot) function in [libtrp] 140
(troeps.type.class-type class) function in [libtrp] 140
(troeps.type.compare type1 type2) — generic in [libtrp] 140
(troeps.type.conceptslot-type conceptslot) function in [libtrp]..... 140
(troeps.type.concept-type concept) function in [libtrp]..... 140

(tr-path-p anything) function in [libtrp] 90
(tr-print-report base) function in [libtrp]..... 17
(tr-print-when-created optional: boolean) macro in [libtrl].....129
(tr-rem-annotation entity label) function in [libtrp] 75
(tr-rem-classslot-assign classslot paths) function in [libtrp] 90
(tr-rem-classslot-filters classslot filter) function in [libtrp]..... 91
(tr-rem-classslot-method classslot method) function in [libtrp] 93
(tr-rem-concept base concept) function in [libtrp]..... 16
(tr-rem-conceptslot-method conceptslot method) function in [libtrp]..... 92
(tr-rem-import base base2) function in [libtrp] 16
(tr-rem-instance-file base filename) function in [libtrp] 17
(tr-rem-lexicon base filename) function in [libtrp]..... 17
(tr-rem-module base modulename) function in [libtrp] 17
(tr-rem-slot-annotation object slotname label) function in [libtrp]..... 74
(tr-rem-type base type) function in [libtrp] 16
(tr-rem-value object slotname value rest: views) function in [libtrp]..... 71
(tr-rem-viewpoint base viewpointname) function in [libtrp]..... 15
(trropes.http.link entity) function in [libtrh].....118
(tr-save-base base) function in [libtrl].....130
(tr-save-base-as-c base optional: fileout)— function in [libtrl].....130
(tr-save-base-as-html base dirbase) function in [libtrh] 117
(tr-save-base-as-talk base optional: fileout) function in [libtrl].....130
(tr-save-base-as-xml base optional: fileout) function in [libtrl] 130
(tr-save-lexicon filename) function in [libtrp]..... 79
(tr-set-annotation entity label annotation) function in [libtrp]..... 74
(tr-set-classslot-card classslot interval) function in [libtrp]..... 53
(tr-set-classslot-condition classslot constraint-predicate)— function in [libtrp]..... 96
(tr-set-classslot-default classslot value) function in [libtrp] 88
(tr-set-classslot-domain classslot domain) function in [libtrp]..... 52
(tr-set-classslot-except classslot domain) function in [libtrp]..... 52
(tr-set-classslot-intervals classslot intervals) function in [libtrp] 52
(tr-set-classslot-type classslot type) function in [libtrp]..... 52
(tr-set-slot-annotation object slotname label annotation) function in [libtrp] 74
(tr-set-value object slotname value rest: views) function in [libtrp] 70
(tr-type-cardinal type value) function in [libtrp].....139
(tr-type-disimilarity type value1 value2) function in [libtrp] 139
(tr-type-equal-p type value1 value2) function in [libtrp] 139
(tr-type-member-p type value) function in [libtrp].....139
(tr-type-p anything) macro in [libtrp] 139
(tr-type-predecessor type value) function in [libtrp] 139
(tr-type-read-value type string) function in [libtrp] 140
(tr-type-string type value) generic in [libtrp].....139
(tr-type-successor type value) function in [libtrp].....139
(tr-type-superior-p type value1 value2) function in [libtrp] 139
(tr-unload-instances base) function in [libtrl].....130
(tr-unset-classslot-default classslot) function in [libtrp]..... 88

(tr-unset-value object slotname rest: views) function in [libtrp]..... 70
(tr-version) function in [libtrp]..... 10
edit-p global-variable in [libtrh].....118
revision-p global-variable in [libtrh].....118