The AMOS Project

IST-2001-34717

The Internal Query Language Design

Deliverable D3

CLIP Technical Report CLIP 1/03.1

Responsible person: Manuel Carro, Technical University of Madrid (mcarro@fi.upm.es)

Release history

Number	*Author*	*Comments*
0.0	Manuel Carro	First draft
0.1	Manuel Carro, Jesús Correas, José Gómez	Added interfaces
0.2	Manuel Carro, Jesús Correas, José Gómez	Main body rewritten, added figures

Abstract

We describe the internal query language used in the communication among the interface of AMOS, the matching engine, and the implementation of the ontology. While a great part of the definition of this query language deals with consults and updates of an instance of the ontology, we do not commit to a given mapping onto a database scheme, which will be defined later in the project. Instead, we rely on high-level primitives and we adopt a semantics similar to that of logic programming semantics, with which interfaces can be expressed concisely and at a high-level.

Since the tool is also supposed to be interactive (the user should be able to perform a refinement of the search), part of the query language deals also with performing searches. Finally, part of the functionality of the tool is related with updates of the contents of the database: it must be possible to enter and validate new data about software projects, and thus the interface also caters for the possibility of database updates.

Contents

1 Introduction

AMOS is a tool aimed at generating sets of Open Source Code software packages (an *assembly*) from a database of descriptions thereof. Packages are generated after a search, which is based on:

- **User requirements**, stating which *capabilities* are needed, expressed by means of a set of words from a fixed, finite, and well defined set of terms, and
- The **descriptions** of software packages, using terms from the aforementioned set, which reflect (among other characteristics) which capabilities are needed by a package, and which capabilities are offered by it.

This approach, which can be termed as *indexical retrieval*, following the classification in [MMM95], has several interesting properties. For example, its retrieval rate is 100% in the sense that elements sought for are always reached if present in the (finite) database. Big databases may lead to a combinatorial explosion in the size of the search tree and in the number of retrieved packages, which suggests using:

Ordering heuristics in order to try to lead the search to the better solutions faster. These should be expressible by the user, and maybe composable.

Incomplete searches which do not necessarily impose a goodness measure on the selection at each point, but which may remove some (non-promising) branches at each search node, therefore reducing the effective size of the search tree.

Interactive user-driven refinement with which the user states first a general query and, depending on the number and quality of the results this query yields, a new query (maybe more, maybe less general) is created and explored.

While such a tool could in principle be used as a command-line program, a graphical user interface is a must nowadays. Since access to AMOS descriptions and recovery facilities should be made public, a WWW-based interface offers platform-independent and world-wide reachability with a relatively low deployment effort. We will therefore, and as planned in the project workplan, aim at a WWW interaction. The internal query language, however, will not be tied to this type of interface and can, in principle, be used with other types of clients (e.g., standalone Java clients which access to an AMOS server running remotely).

The description of the code resources (packages) obeys to instances of the ontology for Open Software Code designed as part of this project [Daf02]. This ontology is to be ultimately mapped on database [1] schematas for its implementation and deployment. As the access schemes might differ according on the underlying technology, we have isolated representation details and provided a high level interface which is notwithstanding enough as to fulfill the needs of the matching engine and the interface generator.

The relationships among the interface, the matching engine, and the database containing the descriptions of packages is shown in Figure 1. The arrows are meant to represent usage of external interface primitives, classified according to their functionality. The set of primitives offered by each of the main modules make up the general *Query Interface*, which implements methods to perform:

[1]Note that, on purpose, we are not committing to any vendor, implementation, or database technology, as we want the ontology design to be general enough as to be easily adaptable for future developments.

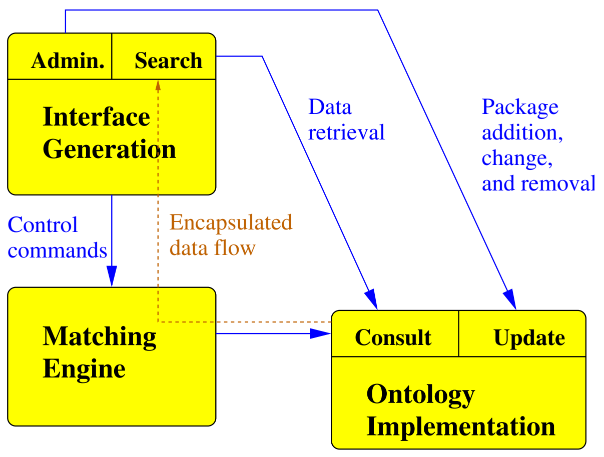

Figure 1: Interface, matching engine, and database

1. Consults to the database (to be made by the matching engine, by the search interface, and by the administrative interface),

2. Updates to the database (performed by the administrative interface),

3. And control of the search process (performed by the search interface).

In the rest of this document we will first describe the guidelines which governed the design of the different operations of the query language and its general functionality, tackling in order the three points above. Full details are available in the appendices of this document.

2 Creating and Acessing the Ontology

The ontology definition states at a high level which data is necessary to describe an open source package, and includes related information as to which license is applicable, from where it can be downloaded, etc. This description is structured at a conceptual level, i.e., it does not try to determine which fields are key, it does not try to maintain a normal form (e.g., lists are a primitive data type), and it uses *extension* of classes often, much like object-oriented designs do.

It should be noted that while adapting an ontology to be implemented into a relational database requires the usual normalization steps, viewing (and also implementing) the ontology in Prolog — or almost any other related logic-based language— requires much less effort, since lists, records and their composition of records to construct other records, search (in the form of backtraking), and aggregation (e.g., gathering all the results of a search in a single list) are all first-order citizens, which greatly eases design, prototyping, and final implementation. Even more, Prolog can express quite naturally complex dependencies and consistency / integrity constraints which are sometimes stated in ontology descriptions. In fact, when such constraints are present, the language of first-order logic is quite often used, which can in many cases be directly translated into Prolog predicates (as done, for example, in [ACFLGP01] among others) which ensure the internal coherence of the instances of the ontology. Translating ontologies into Prolog programs (and viceversa) is, in fact, a technique used since long ago [Far95] and which is being used in several ontology-related applications, ranging from multisource information fusion [KW02] to ontology retrieval from unstructured sources [SPRS02, Tec].

We will assume, in the design of the interface, that Prolog semantics and capabilities available, since they make the design much clearer and closer to a conceptual implementation, and also because the implementation is to be made in Prolog. This semantics include the bidirectionality of arguments (i.e., procedure arguments can be be used both to construct and to consult a data structure) and the implicit non-determinism implemented as backtracking.

We will sketch in this section the general approach to constructing and consulting instances classes as defined in the ontology. A more detailed description of the different primitives, based on the ontology described in [Daf02], is to be found in Appendix A to H. We will do the same for the interface to add class instances to the database and to retrieve them (Section 2.2), and to perform searches for packages (Section 3). All these interfaces are documented more in depth in the Appendices.

2.1 Ontology Construction and Traversal Primitives

The interface reflects shape of the ontology, capturing the information in it at the same level as the ontology (Figure 2). The operations allow both building (and updating) instances of a class and consulting parts of the class itself. The general form of the operations to perform that is

```
classname_fieldname(Object, Value, NewObject)
```

It can be used either to:

- **Construct** a part of an instance of the ontology using a (new) value for a given field, and returns the new piece of the ontology, as in

  ```
  asset_author(Asset, "The Clip Lab", NewAsset)
  ```

- **Retrieve** the value of a field, as in

  ```
  asset_author(Asset, Author, Asset)
  ```

  (note how the first and third variables have the same name; using a fresh, new variable for the third argument would yield the same effect).

- Or **check** the value of some field in a given class instance:

  ```
  asset_author(Asset, "The Clip Lab", Asset)
  ```

  This last query will succeed if the value of the `author` field in the `asset` class unifies with `"The Clip Lab"`, and will fail (without changing the ontology) otherwise.

All modules implementing access to classes have additionally class constructors (such as as `asset(A)`), which either instantiate `A` to an (empty) `asset` class skeleton, or check whether `A` is bound to an `asset` class, and general `class_update/3` calls which allow updating in a single call a series of fields in the object. These updates are all backtrackable, so that declarativeness is not lost.

Basic datatypes, such as strings and numbers, are all built-in in Ciao Prolog. Complex types, such as lists, records (without restrictions as to what the record components are) are also available as primitive datatypes with automatic memory management. This allows building complex data structures with a minimal effort and handling them without having to take care of memory allocation / deallocation.

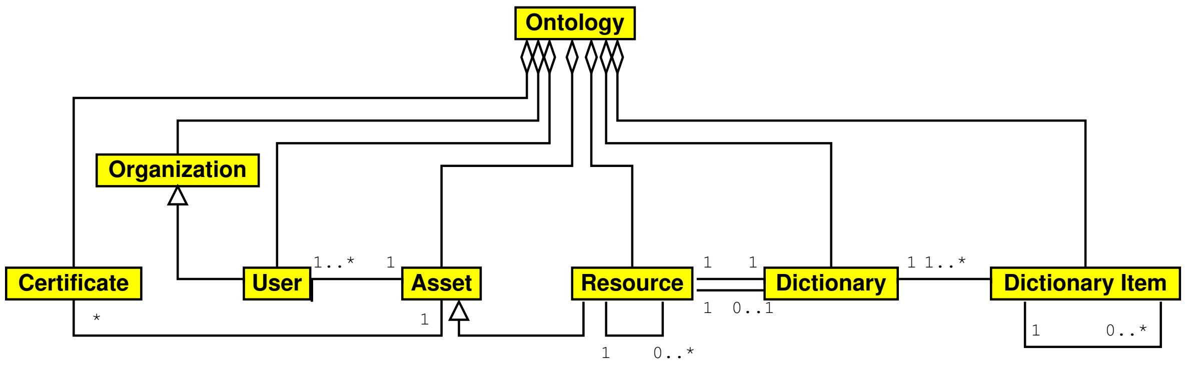

Figure 2: Ontology shape in an UML-like representation

2.2 Storing and Retrieving Data from the Database

The database interface tries to isolate low-level details from the programmer, but, on the other hand, should provide a set of operations which can be efficiently implemented. The database interface works directly with the data structures used to generate ontology instances (Section 2.1), and, at the same time, is aware of the existence of *database identifiers*, opaque data structures whose implementation is dependent on the underlying database technology.

Retrieving class instances from the database is implemented by a series of primitives of the form

```
classname_description(Description, Key)
```

where `Key` is an identifier for the class instance[2] which can be the value of a field of `class`, or a key generated by the database implementation. In any case, it is a value not supposed to be changed or generated directly by users of the database acccess module. As an example,

```
resource_description(Description, Key)
```

returns in `Description` a `resource` instance (the same data structure which was generated using the operations in Section 2.1), which can then be traversed, dissassembled, and changed. `Key` will be an identifier which may or may not be the same as the value of the `AssetName` field of the class `asset`.

There is a special case where it is advisable to have a special operation in order to improve perfomance as much as possible. It is desirable to speed up the access to search-related data (e.g., what a resource offers, and what a resource demands). Therefore, specialized primitives which only retrieve items pertaining to the search procedure are introduced

```
resource_needs(Key, ListOfNeededCapabilities)

resource_provides(Key, ListOfProvidedCapabilities)
```

which, for a given `Key`, return the list of capabilities needed by this package (i.e., a list containing the dictionary terms as stated by the `requires` field in the `resource` class), and the list of provided capabilities, as stated in the `identificationTags` field of the `resource` class. In addition to that, the list of capabilities can also be accessed using directly the resource data structure, as in

[2]Having unique identifiers is not mandatory, but it is in many cases helpful to speed up searches.

```
resource_needs(Resource, ListOfNeededCapabilities)
```

```
resource_provides(Resource, ListOfProvidedCapabilities)
```

In both approaches, the returned result will be in the format more useful to perform searches (e.g., lists of terms). This basic search interface may be eventually updated with more operations which retrieve quickly information needed to implement different search heuristics, for example. The database implementation should also ensure a steadfast implementation of these operations, even if this causes redundancy of information in the ontology implementation. In such case, the database interface is of course responsible for keeping the coherence of the database.

In order to abide by Prolog semantics, all the consult operations can both check and generate answers on backtracking, when called with uninstantiated (or, in general, insufficiently instantiated) variables. For example, a call such as

```
resource_description(_Unneeded, Key)
```

would instantiate `Key` on backtracking to all the resource keys in the database. These can easily be retrieved in a list if needed for further processing (this applies, in general, to all the solutions generated on backtracking). Therefore, special operations which return lists of, e.g., all the author names which have some asset in the database, are not really needed, although they can very easily be added to the interface without breaking its functionality:

```
resource_keys(Keys):-
    setof(ThisKey, resource_description(_, ThisKey), Keys).
```

Needless to say, the previous primitive operations can always be combined to make higher-level, specialized ontology- and database-related procedures, if found useful during the development of the project.

Database updates are assumed to be much less frequent than database consults, and usually under the control of knowledgeable users. Therefore, the update primitives can take a more basic form: using the `resource` class as example

```
resource_update(ResourceKey, NewResource)
```

when called with `ResourceKey` instantiated to a valid key (which should have been previously obtained from the database) it will update the resource information associated with `ResourceKey` to be `NewResource`; the old information is removed from the database. `NewResource` must be a ground term, i.e., a term which does not contain free variables, in order to ensure a proper semantics. Note that this restriction is not necessary, or even convenient, when performing retrievals.

If `resource_update/2` is called with a free variable in the `ResourceKey` argument, the `NewResource` is added to the database, if it does not exist yet, and a new identifier is generated for it. If `NewResource` exists already in the database, `ResourceKey` is instantiated to the associated key. The restriction that no variables can appear in the resource data structure avoids free variables to be present in the database (thus easing its implementation) and also allows a simpler semantics to the database updates.

Permanent removal of information stored in the database is achieved with any of the two following operations:

```
resource_delete(ResourceKey)

resource_delete(Resource)
```

which, again, need `Resource` to be ground. If `Resource` is present in the database (c.f., `ResourceKey` is the key of a resource), the associated information is removed. Otherwise, the database does not change. Removing an item from the database is, of course, a potentially dangerous operation: in order to maintain the database consistency, data which is only referenced by the item being deleted might be removed as well (depending on the minimum cardinality with which the referenced objects appears in their referrers).

2.3 Description of the Ontology

There is a part of the ontology implementation (Appendix A) which cannot be updated by the user, and which has to be changed, if needed, at implementation level. It gives information about the ontology version, author, component classes, etc. Changing this only makes sense if the ontology evolves, which necessarily also involves modifications in the code of some module.

3 The Query Interface

The query interface will receive queries built according to the user preferences stated via the WWW interface, and it will start a search against the information stored in the database. As shown in Figure 1, the matching engine has access to the implementation of the ontology through the database interface. The need to perform involved searches makes it sensible to isolate this facility on a module of its own, with which the interface communicates through a well-defined interface. Having in mind that the interface should offer the possibility of progressively showing more solutions to a query, the basic functions of the search interface are returning (more) solutions to a query and deciding when the solutions have been exhausted.

We will put together what is to be sought for in a *query abstraction*: a datatype which encodes in a first-order, self-contained structure the terms to search for, and also the state of the search (i.e., at which point the search was interrupted, and where it should restart). The basic operations are:

```
build_query(Terms, Heuristic, IncludePacks, ExcludePacks, Query)

make_query(Query, MaxSols, Results, NextQuery)
```

The first operation constructs a new query out of the user selection performed via the WWW interface: the heuristic to follow, the capabilities to search for, packages which have been selected by the user as interesting or advantegeous,[3] and packages which are **not** desired in a final solution. `Results` is a list of at most `MaxSols` solutions, containing each at least a summary of the packages returned by the search procedure. `NextQuery` returns a first-order, self-contained version of the initial query which has been updated to reflect where the search has stopped, so that the next `make_query/4` using `NextQuery` as first argument can restart the search precisely where it was left by the previous `make_query/4`. The query has (finitely) failed (i.e., there a are no [more] solutions) if `Results` is the empty list (and further calls to `make_query/4` will not return any more answers, assuming that the database contents have not changed meanwhile).

[3] This helps to cut down search, as these are usually pakages the user knows to fulfill some of the desired characteriscs.

For brevity, each of the solutions returned in `Results` summarize the total information associated to that solution. This makes it easy to represent a set of solutions in a single WWW page, which the user can quickly glance through. Clicking on one solution will retrieve more information about the packages which make up that solution. In order to implement this information expansion, an operation

```
expand_solution(Solution, Packages, Fulfilled, Flooded)
```

is implemented, which receives a `Solution` (any of the terms in the list of solutions `Results`) and reproduces the part of the search which lead to that solution, gathering more informacion. Due to properties of the search being performed, the order in which packages are retrieved is not important to reproduce faithfully all the intermediate results of a search. Therefore, only the set of packages returned by each of the solutions in `Results` is needed. The more relevant data about the search is the list of capabilities which were needed (either because they were initially requested by the user, or because intermediate chaining needed it) and which were satisfied or not. More information about each of these items (e.g., what is their semantics) can be obtained through the database interface.

Additionally, and as a help for refining queries in a process in which the user interacts with the system, a series of operations to manipulate queries are provided. All of them can be reformulated based on the bidirectionality of `build_query/5` and on standard Prolog list processing, but they are included because of their usefulness:

```
query_add_search_terms(Query, Terms, NewQuery)

query_remove_search_terms(Query, Terms, NewQuery)

query_add_include_packages(Query, Packages, NewQuery)

query_remove_include_packages(Query, Packages, NewQuery)

query_add_exclude_packages(Query, Packages, NewQuery)

query_remove_exclude_packages(Query, Packages, NewQuery)
```

4 Conclusions

We have presented the design of the query language internally used by the tools in the AMOS project. This language casts user requests onto calls to the database implementing the ontology, calls to actually build and traverse class instances, and calls to start queries against the ontology. The operations are designed to be as high-level as possible (paired with the abstraction level and possibilities of Prolog), and to be as close as possible to the conceptual level of the ontology.

Notwithstanding, attention has been paid to efficiency issues in the places where this can be a concern, namely, in the operations related to the search process.

A Information About the Ontology

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 12:5:52 CET)

The ontology module acts as a top-level declaration of external properties of the ontology. It allows **consulting** data such as version, author name and email, and names of other classes the ontology is made of, but it does not give direct access or reexports primitives implementing operations on the subclasses. If the ontology evolves, the changes will be reflected in the ontology access implementation and in the data declared by this interface.

All the operations, but the `ontology/1` constructive type declaration, must use an already-made ontology. E.g.,

```
?- ontology(_O), ontology_author(_O, E).

E = "Carlo Daffara" ? ;
```

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

A.1 Usage and interface (`ontology_access`)

- **Library usage:**

  `:- use_module(library(ontology_access)).`

- **Exports:**
  - *Predicates:*

    `ontology_name/2`, `ontology_version/2`, `ontology_status/2`, `ontology_author/2`, `ontology_authoremail/2`, `ontology_affiliation/2`, `ontology_description/2`, `ontology_comment/2`, `ontology_language/2`, `ontology_metadata/3`.
  - *Regular Types:*

    `ontology/1`.

- **Other modules used:**
  - *Application modules:*

    `basic_types`.
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

A.2 Documentation on exports (`ontology_access`)

ontology_name/2: PREDICATE

Usage: `ontology_name(Ont,Name)`

- *Description:* `Name` is the name of the ontology `Ont`.
- *The following properties should hold at call time:*

  `Ont` is currently a term which is not a free variable. (`nonvar/1`)

  `Ont` is an ontology to store information about open source projects. (`ontology/1`)

  `Name` is a string (a list of character codes). (`string/1`)

ontology_version/2: PREDICATE

Usage: `ontology_version(Ont,Version)`

- *Description:* `Version` is the version of the ontology `Ont`.
- *The following properties should hold at call time:*

  `Ont` is currently a term which is not a free variable. (`nonvar/1`)

  `Ont` is an ontology to store information about open source projects. (`ontology/1`)

  `Version` is a string (a list of character codes). (`string/1`)

ontology_status/2: PREDICATE

Usage: `ontology_status(Ont,Status)`

- *Description:* `Status` is the status of the ontology `Ont`.
- *The following properties should hold at call time:*

`Ont` is currently a term which is not a free variable.	(`nonvar/1`)
`Ont` is an ontology to store information about open source projects.	(`ontology/1`)
`Status` is a string (a list of character codes).	(`string/1`)

ontology_author/2: PREDICATE

Usage: `ontology_author(Ont,Author)`

- *Description:* `Author` is the author name of the ontology `Ont`.
- *The following properties should hold at call time:*

`Ont` is currently a term which is not a free variable.	(`nonvar/1`)
`Ont` is an ontology to store information about open source projects.	(`ontology/1`)
`Author` is a string (a list of character codes).	(`string/1`)

ontology_authoremail/2: PREDICATE

Usage: `ontology_authoremail(Ont,Email)`

- *Description:* `Email` is the author email of the ontology `Ont`.
- *The following properties should hold at call time:*

`Ont` is currently a term which is not a free variable.	(`nonvar/1`)
`Ont` is an ontology to store information about open source projects.	(`ontology/1`)
`Email` is a string (a list of character codes).	(`string/1`)

ontology_affiliation/2: PREDICATE

Usage: `ontology_affiliation(Ont,Affiliation)`

- *Description:* `Affiliation` is the affiliation of the author of the ontology `Ont`.
- *The following properties should hold at call time:*

`Ont` is currently a term which is not a free variable.	(`nonvar/1`)
`Ont` is an ontology to store information about open source projects.	(`ontology/1`)
`Affiliation` is a string (a list of character codes).	(`string/1`)

ontology_description/2: PREDICATE

Usage: `ontology_description(Ont,Description)`

- *Description:* `Description` is a short description of the ontology `Ont`.

– *The following properties should hold at call time:*
`Ont` is currently a term which is not a free variable. (`nonvar/1`)
`Ont` is an ontology to store information about open source projects. (`ontology/1`)
`Description` is a string (a list of character codes). (`string/1`)

ontology_comment/2: PREDICATE

Usage: `ontology_comment(Ont,Comment)`

– *Description:* `Comment` describes the current status of the ontology `Ont`.
– *The following properties should hold at call time:*
`Ont` is currently a term which is not a free variable. (`nonvar/1`)
`Ont` is an ontology to store information about open source projects. (`ontology/1`)
`Comment` is a string (a list of character codes). (`string/1`)

ontology_language/2: PREDICATE

Usage: `ontology_language(Ont,Language)`

– *Description:* `Language` is the language name of the ontology `Ont`.
– *The following properties should hold at call time:*
`Ont` is currently a term which is not a free variable. (`nonvar/1`)
`Ont` is an ontology to store information about open source projects. (`ontology/1`)
`Language` is a valid language descriptor. It consists of a pair `lang(String, LangCode)`, where `String` is a human-readable language name and `LangCode` is an atom corresponding to the (standard) coding of languages:

```
language(lang("English",'EN')).
language(lang("Spanish",'SP')).
language(lang("French",'FR')).
language(lang("German",'DE')).
```

(`language/1`)

ontology_metadata/3: PREDICATE

Usage: `ontology_metadata(Ont,Parent,Children)`

– *Description:* `Parent` is the parent class of `Ont` (or 'none' if `Ont` does not have parents). `Children` is the list of classes `Ont` has as children
– *The following properties should hold at call time:*
`Ont` is currently a term which is not a free variable. (`nonvar/1`)
`Ont` is an ontology to store information about open source projects. (`ontology/1`)
`Parent` is an atom. (`atm/1`)
`Children` is a list of `atm`s. (`list/2`)

ontology/1: REGTYPE

Usage: `ontology(Ont)`

– *Description:* `Ont` is an ontology to store information about open source projects.

A.3 Version/Change Log (`ontology_access`)

Version 0.1#1 (2003/2/14, 12:5:52 CET) Added module explanation, updated arities, now working (Manuel Carro)

Version 0.1 (2003/2/13, 13:30:10 CET) First version (Manuel Carro)

B Creating and Consulting Assets

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#4 (2003/2/20, 1:30:34 CET)

Assets are the basic abstraction resources (e.g., packages, documentation, etc.) are built on. An asset contains information pertaining all the relevant information of a resource, but the dependencies themselves. This is so because there might be resources which do not have explicit dependencies to search for (e.g., pieces of work which are not ultimately intended to be compiled, such as licenses). In order to cater for the existence of these entities, the *asset* abstraction reflects interesting characteristics of such non-software (but anyway interesting and needed) packages.

The operations herein included are aimed at creating and consulting instances of the *asset* class. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

B.1 Usage and interface (`asset_access`)

- **Library usage:**

  `:- use_module(library(asset_access)).`

- **Exports:**

  – *Predicates:*

    `asset_name/3`, `asset_author/3`, `asset_maintained_by/3`, `asset_homepage/3`, `asset_download_page/3`, `asset_contact/3`, `asset_license/3`, `asset_license_URL/3`, `asset_version/3`, `asset_references/3`, `asset_additional_constraints/3`, `asset_additional_freedom/3`, `asset_description/3`, `asset_creation_date/3`, `asset_submitted_by/3`, `asset_submission_date/3`, `asset_environment/3`, `asset_cost/3`, `asset_security_classification/3`, `asset_certification/3`, `asset_package_signature/3`, `asset_check/2`, `asset_update/3`.

  – *Regular Types:*

    `asset/1`, `asset_field/1`.

- **Other modules used:**

  – *Application modules:*

    `user_access`, `basic_types`.

  – *System library modules:*

    `sort`, `aggregates`.

  – *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

B.2 Documentation on exports (`asset_access`)

asset_name/3: PREDICATE

Usage: `asset_name(Asset,Name,NewAsset)`

– *Description:* `Name` is the name (identifier) of `NewAsset`. It should be unique within the system, and its presence is required.

– *The following properties should hold at call time:*

`Asset` is an asset. (`asset/1`)

`Name` is a string (a list of character codes). (`string/1`)

`NewAsset` is an asset. (`asset/1`)

asset_author/3: PREDICATE

Usage: `asset_author(Asset,Authors,NewAsset)`

- *Description:* `Authors` is the list of authors of `NewAsset`. Its presence is required, an there must be at least one author.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Authors` is a list of `users`. (`list/2`)

  `NewAsset` is an asset. (`asset/1`)

asset_maintained_by/3: PREDICATE

Usage: `asset_maintained_by(Asset,Maintainers,NewAsset)`

- *Description:* `Maintainers` is the list of (current) maintainers of `NewAsset`. It may be empty.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Maintainers` is a list of `users`. (`list/2`)

  `NewAsset` is an asset. (`asset/1`)

asset_homepage/3: PREDICATE

Usage: `asset_homepage(Asset,Homepage,NewAsset)`

- *Description:* `Homepage` is the identifer of the home page of `NewAsset`.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Homepage` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_download_page/3: PREDICATE

Usage: `asset_download_page(Asset,Download,NewAsset)`

- *Description:* `Download` is an address where `NewAsset` can be downloaded from.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Download` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_contact/3: PREDICATE

Usage: `asset_contact(Asset,Contact,NewAsset)`

- *Description:* `Contact` is the name of a contact person for `NewAsset`.

- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Contact` is a user. (`user/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_license/3: PREDICATE

Usage: `asset_license(Asset,License,NewAsset)`

- *Description:* `License` is the text of the license for `NewAsset`. Its presence is required.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `License` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_license_URL/3: PREDICATE

Usage: `asset_license_URL(Asset,URL,NewAsset)`

- *Description:* `URL` is an address where the license of `NewAsset` can be downloaded from.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `URL` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_version/3: PREDICATE

Usage: `asset_version(Asset,Version,NewAsset)`

- *Description:* `Version` is the version of the asset `NewAsset`.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Version` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_references/3: PREDICATE

Usage: `asset_references(Asset,References,NewAsset)`

- *Description:* `References` is a set of references concerning `NewAsset`.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `References` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_additional_constraints/3: PREDICATE

Usage: `asset_additional_constraints(Asset,Constraints,NewAsset)`

- *Description:* `Constraints` is a description of constraints to be taken into account when using `NewAsset`.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Constraints` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_additional_freedom/3: PREDICATE

Usage: `asset_additional_freedom(Asset,Freedom,NewAsset)`

- *Description:* `Name` is a description of additional properties of the license of `NewAsset`.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Freedom` is a string (a list of character codes). (`string/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_description/3: PREDICATE

Usage: `asset_description(Asset,Description,NewAsset)`

- *Description:* `Description` is a textual description of the asset `NewAsset`. Its presence is required.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Description` is a list of `string`s. (`list/2`)
  `NewAsset` is an asset. (`asset/1`)

asset_creation_date/3: PREDICATE

Usage: `asset_creation_date(Asset,Date,NewAsset)`

- *Description:* `Date` is the date in which `NewAsset` was created.
- *The following properties should hold at call time:*
  `Asset` is an asset. (`asset/1`)
  `Date` is a date. (`date/1`)
  `NewAsset` is an asset. (`asset/1`)

asset_submitted_by/3: PREDICATE

Usage: `asset_submitted_by(Asset,Submitter,NewAsset)`

- *Description:* `Submitter` is the user which submitted `NewAsset`. Its presence is required.

- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Submitter` is a user. (`user/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_submission_date/3: PREDICATE

Usage: `asset_submission_date(Asset,Date,NewAsset)`

- *Description:* `Date` is the date when `NewAsset` was submitted. Its presence is required.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Date` is a date. (`date/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_environment/3: PREDICATE

Usage: `asset_environment(Asset,Environment,NewAsset)`

- *Description:* `Environment` is a textual description of the environment `NewAsset` was designed for or used in.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Environment` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_cost/3: PREDICATE

Usage: `asset_cost(Asset,Cost,NewAsset)`

- *Description:* `Cost` is the assumed cost of `NewAsset`. It will probably be not finally used, but it is included for generality: some assets might require, e.g., an initial cost.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Cost` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_security_classification/3: PREDICATE

Usage: `asset_security_classification(Asset,Class,NewAsset)`

- *Description:* `Class` is the security classification of `NewAsset`.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Class` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_certification/3: PREDICATE

Usage: `asset_certification(Asset,Certification,NewAsset)`

- *Description:* `Certification` is the list of certifications applicable to `NewAsset`. It may be empty.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Certification` is a list of `certification`s. (`list/2`)

  `NewAsset` is an asset. (`asset/1`)

asset_package_signature/3: PREDICATE

Usage: `asset_package_signature(Asset,Signature,NewAsset)`

- *Description:* `Signture` is the signature of the of `NewAsset`; it is included in order to make it possible to check the integrity of the assets.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `Signature` is a string (a list of character codes). (`string/1`)

  `NewAsset` is an asset. (`asset/1`)

asset_check/2: PREDICATE

Usage: `asset_check(Asset,WrongFields)`

- *Description:* The already formed asset `Asset` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields.
- *The following properties should hold at call time:*

  `Asset` is an asset. (`asset/1`)

  `WrongFields` is a list of `asset_fields`s. (`list/2`)

asset_update/3: PREDICATE

Usage: `asset_update(Asset,PairList,NewAsset)`

- *Description:* This operation updates `Asset` to give `NewAsset`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its updated value. `Asset update` is a general call upon which all the rest of the `update` calls rely.

– *The following properties should hold at call time:*

`Asset` is currently a term which is not a free variable. (`nonvar/1`)

`PairList` is currently a term which is not a free variable. (`nonvar/1`)

`Asset` is an asset. (`asset/1`)

`PairList` is a list of `pair`s. (`list/2`)

`NewAsset` is an asset. (`asset/1`)

asset/1: REGTYPE

Usage: `asset(Asset)`

– *Description:* `Asset` is an asset.

asset_field/1: REGTYPE

Usage: `asset_field(Field)`

– *Description:* `Field` is a field of an asset

It is currently defined as:

```
asset_field(name).
asset_field(author).
asset_field(maintainer).
asset_field(homepage).
asset_field(downloadpage).
asset_field(contact).
asset_field(license).
asset_field(licenseurl).
asset_field(version).
asset_field(references).
asset_field(constraints).
asset_field(freedom).
asset_field(assetdescription).
asset_field(creationdate).
asset_field(submittedby).
asset_field(submissiondate).
asset_field(environment).
asset_field(cost).
asset_field(security).
asset_field(certification).
asset_field(signature).
```

B.3 Version/Change Log (`asset_access`)

Version 0.1#4 (2003/2/20, 1:30:34 CET) Ast to Asset (Manuel Carro)

Version 0.1#3 (2003/2/14, 16:25:56 CET) Changed AssetAfter (Manuel Carro)

Version 0.1#2 (2003/2/14, 13:33:59 CET) Added a general asset_update (Manuel Carro)

Version 0.1#1 (2003/2/14, 12:26:36 CET) Added asset_fields (Manuel Carro)

Version 0.1 (2003/2/13, 20:48:26 CET) Initial version, with all interfaces included. (Manuel Carro)

C Handling Information About Organizations

Author(s): Manuel Carro and the CLIP group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 16:27:27 CET)

The class "Organization" abstracts real organizations (companies, universities, or even user groups), and ultimately reflects an user's affiliation. An organization provides a means to locate and identify a single user (which can be thought of as an extension of an organization) or a group of users who are somehow associated and work or produce software under the same umbrella.

The operations herein included are aimed at creating and consulting instances of the *organization* class. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

C.1 Usage and interface (`organization_access`)

- **Library usage:**

  `:- use_module(library(organization_access)).`

- **Exports:**

  - *Predicates:*

    `organization_name/3`, `organization_email/3`, `organization_telephone/3`, `organization_webpage/3`, `organization_notes/3`, `organization_update/3`, `organization_check/2`.

  - *Regular Types:*

    `organization/1`, `organization_field/1`.

- **Other modules used:**

  - *System library modules:*

    `sort`, `aggregates`.

  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

C.2 Documentation on exports (`organization_access`)

organization_name/3: PREDICATE

Usage: `organization_name(Org,Name,NewOrg)`

- *Description:* `Description` is the description of the organization `NewOrg`. There should be a defined `Name` in each organization
- *The following properties should hold at call time:*

  `Org` is an organization. (`organization/1`)

  `Name` is a string (a list of character codes). (`string/1`)

  `NewOrg` is an organization. (`organization/1`)

organization_email/3: PREDICATE

Usage: `organization_email(Org,Emails,NewOrg)`

- *Description:* `Emails` is a list of contact emails for the organization `NewOrg`. `Emails` may be the empty list
- *The following properties should hold at call time:*

  `Org` is an organization. (`organization/1`)

`Emails` is a list of `string`s. (`list/2`)

`NewOrg` is an organization. (`organization/1`)

organization_telephone/3: PREDICATE

Usage: `organization_telephone(Org,Telephones,NewOrg)`

- *Description:* `Telephones` is a list of contact phone/fax number for the organization `NewOrg`. `Telephones` may be the empty list
- *The following properties should hold at call time:*

  `Org` is an organization. (`organization/1`)

  `Telephones` is a list of `string`s. (`list/2`)

  `NewOrg` is an organization. (`organization/1`)

organization_webpage/3: PREDICATE

Usage: `organization_webpage(Org,Webpages,NewOrg)`

- *Description:* `Webpages` is a list of Web pages for the organization `NewOrg`. `Webpages` may be the empty list
- *The following properties should hold at call time:*

  `Org` is an organization. (`organization/1`)

  `Webpages` is a list of `string`s. (`list/2`)

  `NewOrg` is an organization. (`organization/1`)

organization_notes/3: PREDICATE

Usage: `organization_notes(Org,Notes,NewOrg)`

- *Description:* `Notes` is a list of strings (comments regarding the organization `NewOrg`). `Notes` may be the empty list
- *The following properties should hold at call time:*

  `Org` is an organization. (`organization/1`)

  `Notes` is a list of `string`s. (`list/2`)

  `NewOrg` is an organization. (`organization/1`)

organization_update/3: PREDICATE

Usage: `organization_update(Org,PairList,NewOrg)`

- *Description:* It updates `Org` to give `NewOrg`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its updated value. `Organization update` is a general call upon which all the rest of the `update` calls rely.

– *The following properties should hold at call time:*

`Org` is currently a term which is not a free variable.	(`nonvar/1`)
`PairList` is currently a term which is not a free variable.	(`nonvar/1`)
`Org` is an organization.	(`organization/1`)
`PairList` is a list of `pair`s.	(`list/2`)
`NewOrg` is an organization.	(`organization/1`)

organization_check/2: PREDICATE

Usage: `organization_check(Org,WrongFields)`

– *Description:* The already formed organization class `Org` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields.

– *The following properties should hold at call time:*

`Org` is an organization.	(`organization/1`)
`WrongFields` is a list of `organization_fields`s.	(`list/2`)

organization/1: REGTYPE

Usage: `organization(Org)`

– *Description:* `Org` is an organization.

organization_field/1: REGTYPE

Usage: `organization_field(Field)`

– *Description:* `Field` is a field of a dictionary item
It is currently defined as:

```
organization_field(name).
organization_field(email).
organization_field(telephone).
organization_field(webpage).
organization_field(notes).
```

C.3 Version/Change Log (`organization_access`)

Version 0.1#1 (2003/2/14, 16:27:27 CET) OrgAfter to NewOrg (Manuel Carro)

Version 0.1 (2003/2/13, 20:49:29 CET) Initial version, with all interfaces included. (Manuel Carro)

D Creating and Consulting Certificates

Author(s): Manuel Carro and the CLIP group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 16:25:41 CET)

Certificates are means to ensure that the properties stated for a given package hold true, and that the package has not been altered (for example, by changing code so that misfunction is possible). Certificates are mandatory (or, at least, highly recommended) in several application fields (e.g., health care), and can take a range of forms, from fingerprints such as MD5 to digital signatures or, possibly in a future, to proof-carrying code. Certificates are associated to assets, in such a way that a given asset may have different certifications created using different methods or issued by different (third-part) trusted organizations.

The operations herein included are aimed at creating and consulting instances of the *certification* class. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

D.1 Usage and interface (`certification_access`)

- **Library usage:**

  `:- use_module(library(certification_access)).`

- **Exports:**
  - *Predicates:*

    `certification_date/3`, `certification_status/3`, `certification_level/3`, `certification_policy/3`, `certification_reference/3`, `certification_artifact_type/3`, `certification_artifact/3`, `certification_verifier/3`, `certification_comments/3`, `certification_check/2`, `certification_update/3`.
  - *Regular Types:*

    `certification/1`, `certification_field/1`.

- **Other modules used:**
  - *Application modules:*

    `basic_types`.
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

D.2 Documentation on exports (`certification_access`)

certification_date/3: PREDICATE

Usage: `certification_date(Cert,Date,NewCert)`

- *Description:* `Date` is the date in which `NewCert` was issued. It must be present.
- *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)
`Date` is a date.	(`date/1`)
`NewCert` is a certification.	(`certification/1`)

certification_status/3: PREDICATE

Usage: `certification_status(Cert,Status,NewCert)`

- *Description:* `Status` is the status of `NewCert`. It must be present.
- *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)

`Status` is a string (a list of character codes). (`string/1`)
`NewCert` is a certification. (`certification/1`)

certification_level/3: PREDICATE

Usage: `certification_level(Cert,Level,NewCert)`

- *Description:* `Level` is the level of `NewCert`. It must be present.
- *The following properties should hold at call time:*
  `Cert` is a certification. (`certification/1`)
  `Level` is a string (a list of character codes). (`string/1`)
  `NewCert` is a certification. (`certification/1`)

certification_policy/3: PREDICATE

Usage: `certification_policy(Cert,Policy,NewCert)`

- *Description:* `Policy` is the policy of `NewCert`. It must be present.
- *The following properties should hold at call time:*
  `Cert` is a certification. (`certification/1`)
  `Policy` is a string (a list of character codes). (`string/1`)
  `NewCert` is a certification. (`certification/1`)

certification_reference/3: PREDICATE

Usage: `certification_reference(Cert,References,NewCert)`

- *Description:* `References` are references concerning the certification `NewCert`. It must be present.
- *The following properties should hold at call time:*
  `Cert` is a certification. (`certification/1`)
  `References` is a string (a list of character codes). (`string/1`)
  `NewCert` is a certification. (`certification/1`)

certification_artifact_type/3: PREDICATE

Usage: `certification_artifact_type(Cert,Type,NewCert)`

- *Description:* `Type` is the type of `NewCert`. It must be present.
- *The following properties should hold at call time:*
  `Cert` is a certification. (`certification/1`)
  `Type` is a string (a list of character codes). (`string/1`)
  `NewCert` is a certification. (`certification/1`)

certification_artifact/3: PREDICATE

Usage: `certification_artifact(Cert,Artifact,NewCert)`

– *Description:* `Artifact` is the method used to generate the certification in of `NewCert`. Commonly used methods are SHA, MD5 or other hash mark directly extracted from the resource. It must be present.
– *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)
`Artifact` is a string (a list of character codes).	(`string/1`)
`NewCert` is a certification.	(`certification/1`)

certification_verifier/3: PREDICATE

Usage: `certification_verifier(Cert,Verifier,NewCert)`

– *Description:* `Verifier` is a verifier of `NewCert`. It must be present.
– *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)
`Verifier` is a string (a list of character codes).	(`string/1`)
`NewCert` is a certification.	(`certification/1`)

certification_comments/3: PREDICATE

Usage: `certification_comments(Cert,Comments,NewCert)`

– *Description:* `Comments` are general comments regarding the certification `NewCert`. It must be present.
– *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)
`Comments` is a string (a list of character codes).	(`string/1`)
`NewCert` is a certification.	(`certification/1`)

certification_check/2: PREDICATE

Usage: `certification_check(Cert,WrongFields)`

– *Description:* The already formed certificate `Cert` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields.
– *The following properties should hold at call time:*

`Cert` is a certification.	(`certification/1`)
`WrongFields` is a list of `certification_fields`s.	(`list/2`)

certification_update/3: PREDICATE

Usage: `certification_update(Cert,PairList,NewCert)`

- *Description:* It updates `Cert` to give `NewCert`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its update value. `certification_update/3` is a general call upon which all the rest of the `update` calls rely.
- *The following properties should hold at call time:*

`Cert` is currently a term which is not a free variable.	(`nonvar/1`)
`PairList` is currently a term which is not a free variable.	(`nonvar/1`)
`Cert` is a certification.	(`certification/1`)
`PairList` is a list of `pair`s.	(`list/2`)
`NewCert` is a certification.	(`certification/1`)

certification/1: REGTYPE

Usage: `certification(Cert)`

- *Description:* `Cert` is a certification.

certification_field/1: REGTYPE

Usage: `certification_field(Field)`

- *Description:* `Field` is a field of a certification
  It is currently defined as:

```
certification_field(date).
certification_field(status).
certification_field(level).
certification_field(policy).
certification_field(reference).
certification_field(artifact_type).
certification_field(artifact).
certification_field(verifier).
certification_field(comments).
```

D.3 Version/Change Log (`certification_access`)

Version 0.1#1 (2003/2/14, 16:25:41 CET) Changed CertAfter to NewCert (Manuel Carro)

Version 0.1 (2003/2/13, 20:48:55 CET) Initial version, with all interfaces included. (Manuel Carro)

E Handling Information About Resources

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 16:28:1 CET)

A resource is a piece of information the database knows how to search for. Every resource contains, at least, the information (description terms) needed to express what capabilities are provided by the resource, and which other capabilities are needed in order to use it in a software project. Every resource is part of a more general class, the *asset*, which contains additional information about the resource (e.g., where the resource can be located, etc.) which is however not needed (and, in fact, not enough) as to distinguish several resources from each other.

The operations herein included are aimed at creating and consulting instances of the *resource* class. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

E.1 Usage and interface (`resource_access`)

- **Library usage:**

  `:- use_module(library(resource_access)).`

- **Exports:**

  - *Predicates:*

    `resource_size/3`, `resource_identificationTags/3`, `resource_language/3`, `resource_newVersionOf/3`, `resource_uses/3`, `resource_requires/3`, `resource_notes/3`, `resource_certification_status/3`, `resource_asset/3`, `resource_check/2`, `resource_update/3`.

  - *Regular Types:*

    `resource/1`, `resource_field/1`.

- **Other modules used:**

  - *Application modules:*

    `certification_access`, `asset_access`, `dictionary_access`.

  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

E.2 Documentation on exports (`resource_access`)

resource_size/3: PREDICATE

Usage: `resource_size(Res,Size,NewRes)`

- *Description:* `Size` is the (estimated) size of the resource `NewRes`. There should be at least one size in every well-formed resource.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)

  `Size` is an integer. (`int/1`)

  `NewRes` is a resource. (`resource/1`)

resource_identificationTags/3: PREDICATE

Usage: `resource_identificationTags(Res,Tags,NewRes)`

- *Description:* `Tags` is a list of terms which define the resource `NewRes` in terms of the capabilities it provides. `Tags` must exist and it must **not** be an empty list.

– *The following properties should hold at call time:*

`Res` is a resource.	(`resource/1`)
`Tags` is a list of `ditem`s.	(`list/2`)
`NewRes` is a resource.	(`resource/1`)

resource_language/3: PREDICATE

Usage: `resource_language(Res,Language,NewRes)`

– *Description:* `Language` is the programming language used in the implementation of the resource `NewRes`.

– *The following properties should hold at call time:*

`Res` is a resource.	(`resource/1`)
`Language` is one entry of the dictionary.	(`ditem/1`)
`NewRes` is a resource.	(`resource/1`)

resource_newVersionOf/3: PREDICATE

Usage: `resource_newVersionOf(Res,Forerunner,NewRes)`

– *Description:* `Forerunner` is the name of the resource `NewRes` is a new version of.

– *The following properties should hold at call time:*

`Res` is a resource.	(`resource/1`)
`Forerunner` is an atom.	(`atm/1`)
`NewRes` is a resource.	(`resource/1`)

resource_uses/3: PREDICATE

Usage: `resource_uses(Res,Uses,NewRes)`

– *Description:* `Uses` is the list of resources (determined by their unique name) which can be used when working with the resource `NewRes`. They are not intended to be strong requirements, but their presence might help in compiling, linking, etc. `NewRes`. The list of used resources may be empty.

– *The following properties should hold at call time:*

`Res` is a resource.	(`resource/1`)
`Uses` is a list of `resource`s.	(`list/2`)
`NewRes` is a resource.	(`resource/1`)

resource_requires/3: PREDICATE

Usage: `resource_requires(Res,Requires,NewRes)`

– *Description:* `Requires` is the list of capabilities which must be present to fully use the resource `NewRes`. They represent strong requirements which will be taken into account when performing a search.

- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)
  `Requires` is a list of `ditem`s. (`list/2`)
  `NewRes` is a resource. (`resource/1`)

resource_notes/3: PREDICATE

Usage: `resource_notes(Res,Notes,NewRes)`

- *Description:* `Notes` is a list of miscellaneous notes regarding the resource `NewRes`.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)
  `Notes` is a string (a list of character codes). (`string/1`)
  `NewRes` is a resource. (`resource/1`)

resource_certification_status/3: PREDICATE

Usage: `resource_certification_status(Res,Certification,NewRes)`

- *Description:* `Certification` is the list of certificates of the resource `NewRes`.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)
  `Certification` is a list of `certification`s. (`list/2`)
  `NewRes` is a resource. (`resource/1`)

resource_asset/3: PREDICATE

Usage: `resource_asset(Res,Asset,NewRes)`

- *Description:* `Asset` is the (more general) asset the resource `NewRes` defines.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)
  `Asset` is an asset. (`asset/1`)
  `NewRes` is a resource. (`resource/1`)

resource_check/2: PREDICATE

Usage: `resource_check(Res,WrongFields)`

- *Description:* The already formed resource `Res` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields.

– *The following properties should hold at call time:*
`Res` is a resource. (`resource/1`)
`WrongFields` is a list of `resource_field`s. (`list/2`)

resource_update/3: PREDICATE

Usage: `resource_update(Res,PairList,NewRes)`

– *Description:* It updates `Res` to give `NewRes`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its update value. `Asset update` is a general call upon which all the rest of the `update` calls rely.

– *The following properties should hold at call time:*
`Res` is currently a term which is not a free variable. (`nonvar/1`)
`PairList` is currently a term which is not a free variable. (`nonvar/1`)
`Res` is a resource. (`resource/1`)
`PairList` is a list of `pair`s. (`list/2`)
`NewRes` is a resource. (`resource/1`)

resource/1: REGTYPE

Usage: `resource(Res)`

– *Description:* `Res` is a resource.

resource_field/1: REGTYPE

Usage: `resource_field(Field)`

– *Description:* `Field` is a field of a resource definition
It is currently defined as:

```
resource_field(size).
resource_field(identification).
resource_field(language).
resource_field(versionof).
resource_field(uses).
resource_field(requires).
resource_field(notes).
resource_field(cert_status).
resource_field(asset).
```

E.3 Version/Change Log (`resource_access`)

Version 0.1#1 (2003/2/14, 16:28:1 CET) ResAfter to NewRes (Manuel Carro)

Version 0.1 (2003/2/13, 20:49:41 CET) Initial version, with all interfaces included. (Manuel Carro)

F Creating and Consulting Dictionaries

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 16:26:43 CET)

Dictionaries hold the terms to be used both to describe Open Source Packages and to state which capabilities are of interest when performing a search. Each dictionary consists of a set of terms (dictionary items); each of these, in turn, may have synonyms (in order to make programmers and users coming from different knowledge areas to use the same dictionary) and term descriptions (in order to clarify what is the exact meaning attributed to every dictionary term). The dictionary is assumed to grow steadily until a sufficient size has been reached, when it should stabilize.

The operations herein included are aimed at creating and consulting instances of the *dictionary* and *dictionaryItem* classes. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

F.1 Usage and interface (`dictionary_access`)

- **Library usage:**

  `:- use_module(library(dictionary_access)).`

- **Exports:**

  - *Predicates:*

    `ditem_entry/3`, `ditem_synonyms/3`, `ditem_generalization/3`, `ditem_translation/3`, `ditem_check/2`, `ditem_update/3`, `dictionary_entries/3`.

  - *Regular Types:*

    `dictionary/1`, `ditem/1`, `ditem_field/1`.

- **Other modules used:**

  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

F.2 Documentation on exports (`dictionary_access`)

ditem_entry/3: PREDICATE

Usage: `ditem_entry(Item,Entry,NewItem)`

- *Description:* `Entry` is the actual name of the concept stored in `NewItem`. Its presence is required.
- *The following properties should hold at call time:*

`Item` is one entry of the dictionary.	(`ditem/1`)
`Entry` is a string (a list of character codes).	(`string/1`)
`NewItem` is one entry of the dictionary.	(`ditem/1`)

ditem_synonyms/3: PREDICATE

Usage: `ditem_synonyms(Item,Synonyms,NewItem)`

- *Description:* `Synonyms` is the list of synonyms for the name of the concept stored in `NewItem`. It may be an empty list.
- *The following properties should hold at call time:*

`Item` is one entry of the dictionary.	(`ditem/1`)
`Synonyms` is a list of `string`s.	(`list/2`)
`NewItem` is one entry of the dictionary.	(`ditem/1`)

ditem_generalization/3: PREDICATE

Usage: `ditem_generalization(Item,Generalization,NewItem)`

- *Description:* `Generalization` is a list of terms which generalize the meaning of `NewItem`. They are used to broaden the search in order to obtain (more) matches. It may be an empty list.
- *The following properties should hold at call time:*
  `Item` is one entry of the dictionary. (`ditem/1`)
  `Generalization` is a list of `ditem`s. (`list/2`)
  `NewItem` is one entry of the dictionary. (`ditem/1`)

ditem_translation/3: PREDICATE

Usage: `ditem_translation(Item,Translation,NewItem)`

- *Description:* `Translation` is a list of pairs which define what is the translation to different languages of the term stored in `NewItem`.
- *The following properties should hold at call time:*
  `Item` is one entry of the dictionary. (`ditem/1`)
  `Translation` is a list of `trans`s. (`list/2`)
  `NewItem` is one entry of the dictionary. (`ditem/1`)

ditem_check/2: PREDICATE

Usage: `ditem_check(Ditem,WrongFields)`

- *Description:* The already formed dictionary item `Ditem` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields.
- *The following properties should hold at call time:*
  `Ditem` is one entry of the dictionary. (`ditem/1`)
  `WrongFields` is a list of `ditem_field`s. (`list/2`)

ditem_update/3: PREDICATE

Usage: `ditem_update(Ditem,PairList,NewDitem)`

- *Description:* It updates `Ditem` to give `NewDitem`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its update value. `Ditem update` is a general call upon which all the rest of the `update` calls rely.

– *The following properties should hold at call time:*

`Ditem` is currently a term which is not a free variable.	(`nonvar/1`)
`PairList` is currently a term which is not a free variable.	(`nonvar/1`)
`Ditem` is one entry of the dictionary.	(`ditem/1`)
`PairList` is a list of `pair`s.	(`list/2`)
`NewDitem` is one entry of the dictionary.	(`ditem/1`)

dictionary_entries/3: PREDICATE

Usage: `dictionary_entries(Dict,Entries,DictAfter)`

– *Description:* `Entries` is the list of items (dictionary entries, as per the definition above) stored in the dictionary `DictAfter`. There must be at least a term in the dictionary.

– *The following properties should hold at call time:*

`Dict` is a dictionary.	(`dictionary/1`)
`Entries` is a list of `ditem`s.	(`list/2`)
`DictAfter` is a dictionary.	(`dictionary/1`)

dictionary/1: REGTYPE

Usage: `dictionary(Dict)`

– *Description:* `Dict` is a dictionary.

ditem/1: REGTYPE

Usage: `ditem(Item)`

– *Description:* `Item` is one entry of the dictionary.

ditem_field/1: REGTYPE

Usage: `ditem_field(Field)`

– *Description:* `Field` is a field of a dictionary item
It is currently defined as:

```
ditem_field(entry).
ditem_field(synonyms).
ditem_field(generalization).
ditem_field(translation).
```

F.3 Version/Change Log (`dictionary_access`)

Version 0.1#1 (2003/2/14, 16:26:43 CET) Changed ItemAfter to NewItem (Manuel Carro)

Version 0.1 (2003/2/13, 20:49:7 CET) Initial version, with all interfaces included. (Manuel Carro)

G Creating and Consulting User Identifications

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#1 (2003/2/14, 16:28:28 CET)

Users exist in the system because they are responsible or somehow linked to resources. The class *user* expresses the minimal information needed to distinguish among users, by establishing the identity (i.e., their digital signature) which is used to mark the packages the user is responsible for, and the organization the user belongs to. A user by itself can be the unique participant on an organization, in which case the organization would consist of a single individual.

The operations herein included are aimed at creating and consulting instances of the *user* class. They may change (for example, to include more operations) should the ontology evolve in that direction.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, 'Ciao Prolog', NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of `Obj`, and it will fail otherwise.

Fields which return a free variable are supposed to be uninitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

G.1 Usage and interface (`user_access`)

- **Library usage:**

  `:- use_module(library(user_access)).`

- **Exports:**

  - *Predicates:*

    `user_affiliation/3`, `user_package_signature/3`, `user_update/3`, `user_check/2`.

  - *Regular Types:*

    `user_field/1`, `user/1`.

- **Other modules used:**

  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

G.2 Documentation on exports (`user_access`)

user_affiliation/3: PREDICATE

Usage: `user_affiliation(User,Affiliation,NewUser)`

- *Description:* `Affiliation` is the list of organizations the `NewUser` belongs to.
- *The following properties should hold at call time:*

  `User` is a user. (`user/1`)

  `Affiliation` is a list of `organization`s. (`list/2`)

  `NewUser` is a user. (`user/1`)

user_package_signature/3: PREDICATE

Usage: `user_package_signature(User,Signature,NewUser)`

- *Description:* `Signature` is the digital signature with which `NewUser` signs its packages.
- *The following properties should hold at call time:*

  `User` is a user. (`user/1`)

  `Signature` is a string (a list of character codes). (`string/1`)

  `NewUser` is a user. (`user/1`)

user_update/3: PREDICATE

Usage: `user_update(User,PairList,NewUser)`

- *Description:* It updates `User` to give `NewUser`. `PairList` is a list of pairs (see `pair`): dash-separated ground terms `Field-Value`, which are meant to express the name of a field and its update value. `User update` is a general call upon which all the rest of the `update` calls rely.
- *The following properties should hold at call time:*

`User` is currently a term which is not a free variable.	(`nonvar/1`)
`PairList` is currently a term which is not a free variable.	(`nonvar/1`)
`User` is a user.	(`user/1`)
`PairList` is a list of `pair`s.	(`list/2`)
`NewUser` is a user.	(`user/1`)

user_check/2: PREDICATE

Usage: `user_check(User,WrongFields)`

- *Description:* The already formed user `User` is checked against the constraints specified in the ontology definition. Fields whose associated value at entry is a **free** variable and whose minimal cardinality is zero are substituted by a `null` value (see the regular type `null/1`). Fields whose associated value at entry is a **free** variable and whose minimal cardinality is greater than zero are reported as having a wrong value in the variable `WrongFields`. Finally, fields whose associated value at entry is not a variable, and whose type does not match the one specified in the ontology, are also reported as wrong fields. .
- *The following properties should hold at call time:*

`User` is a user.	(`user/1`)
`WrongFields` is a list of `user_field`s.	(`list/2`)

user_field/1: REGTYPE

Usage: `user_field(Field)`

- *Description:* `Field` is a field of a user instance
  It is currently defined as:

```
user_field(affiliation).
user_field(signature).
```

user/1: REGTYPE

Usage: `user(User)`

- *Description:* `User` is a user.

G.3 Version/Change Log (`user_access`)

Version 0.1#1 (2003/2/14, 16:28:28 CET) UserAfter to NewUser (Manuel Carro)

Version 0.1 (2003/2/13, 20:49:56 CET) Initial version, with all interfaces included. (Manuel Carro)

H Basic types

Author(s): Manuel Carro and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1 (2003/2/13, 20:48:43 CET)

This module contains basic types used in the database access and ontology handling.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

H.1 Usage and interface (`basic_types`)

- **Library usage:**

  `:- use_module(library(basic_types)).`
- **Exports:**
  - *Regular Types:*

    `date/1`, `language/1`, `null/1`.
- **Other modules used:**
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

H.2 Documentation on exports (`basic_types`)

date/1: REGTYPE

Usage: `date(Date)`

– *Description:* `Date` is a date.

language/1: REGTYPE

Usage: `language(LangDesc)`

– *Description:* `LangDesc` is a valid language descriptor. It consists of a pair `lang(String, LangCode)`, where `String` is a human-readable language name and `LangCode` is an atom corresponding to the (standard) coding of languages:

```
language(lang("English",'EN')).
language(lang("Spanish",'SP')).
language(lang("French",'FR')).
language(lang("German",'DE')).
```

null/1: REGTYPE

Usage: `null(Null)`

- *Description:* `Null` is an atom recognized as the `null` value by the database interface.

H.3 Version/Change Log (`basic_types`)

Version 0.1 (2003/2/13, 20:48:43 CET) Initial version, with all interfaces included. (Manuel Carro)

I Top-Level matching engine interface

Author(s): Jose Manuel Gomez, Manuel Carro, and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

This module provides an interface for building and posting search queries to a database containing descriptions of open source code packages. The interface operations are primarily aimed at creating a bridge between a **WWW interface** (or, in general, any end user) and the **ontology** implementation, offering a series of primitives to connect both ends.

Building means constructing a term which states which capabilities are of interest (or of no interest). Heuristics to be used in order to direct the search towards a more promising answer without exploring the whole of the search space can be stated. Additionally, operations are provided to manage the query as an opaque data structure, in order, e.g., to add / remove search terms from it.

Posting the query refers to actually sending it to the matching engine. Each query posted specifies the number of solutions required (useful to generate web pages) and it returns a **new query** expressing the state of the search after the last solution was found. Posting this new query again will restart the search where it was left before.

In this module, and unless otherwise specified, operations of the form `operation_name(Arg, Value, NewArg)` can be used either:

- To update the relevant field of `Arg` to `Value` and leave the result in `NewArg`, e.g.,

  `class_field(Obj, \"Ciao Prolog\", NewObj)`

  (for some `class` within the ontology, and some `field` in the `class`) leaving the fields other than `field` unchanged,

- To retrieve in `Var` the value of the relevant field without changing it, with, e.g., `class_field(_, Var, Obj)` or `class_field(Obj, Var, Obj)`, where `Var` is a free variable at the moment of call, or

- To check whether some field has a given value without changing it, with, e.g. `class_field(_, Value, Obj)` or `class_field(Obj, Value, Obj)`, where `Value` is bound to a non-variable term at the moment of the call. The call will succeed if `Value` is the same (or at least unifiable) with what is stored in the corresponding field of @ObjArg, and it will fail otherwise.

Fields which return a free variable are supposed to be unitialized (which is incorrect for fields of well-formed class instances which need the presence of a certain field). All fields are uninitialized upon creation of a new class instance.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

I.1 Usage and interface (`matching_access`)

- **Library usage:**

  `:- use_module(library(matching_access)).`

- **Exports:**
  - *Predicates:*

    `build_query/5`, `make_query/4`, `expand_solution/4`, `query_add_search_terms/3`, `query_remove_search_terms/3`, `query_add_include_packages/3`, `query_remove_include_packages/3`, `query_add_exclude_packages/3`, `query_remove_exclude_packages/3`.
  - *Regular Types:*

    `heuristic/1`, `solution/1`, `query/1`, `results/1`.

- **Other modules used:**
  - *Application modules:*

    `../DataBase/dictionary_access`.
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

I.2 Documentation on exports (`matching_access`)

build_query/5: PREDICATE

Usage: `build_query(Terms,Heuristic,IncludePacks,ExcludePacks,Query)`

- *Description:* `Terms` is a a list of dictionary terms to be searched for, describing the desired capabilities. `Heuristic` is the heuristic to follow during the search. `IncludePacks` is a list of packages which have been selected by the user as interesting or advantageous and which **must** be included in the final solution. `ExcludePacks` are packages which are not desired in a final solution, and `Query` is a query including the all these terms and which will be used to perform the search.
- *The following properties should hold at call time:*

  `Terms` is a list of `search_terms`. (`list/2`)

  `Heuristic` is an atom. (`atm/1`)

  `IncludePacks` is a list of `atms`. (`list/2`)

  `ExcludePacks` is a list of `atms`. (`list/2`)

  `Query` is a query. (`query/1`)

make_query/4: PREDICATE

Usage: `make_query(Query,MaxSols,Results,NextQuery)`

– *Description:* `Query` is the query that currently guides the search, `Results` is a list of at most `MaxSols` solutions, and `NextQuery` reflects where the current search has stopped so that `make_query/4` can restart it at that point. If `Results` is an empty list, then no solutions have been found (and further calls to `make_query/4` with `NextQuery` will not yield any more solutions).

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable.	(`nonvar/1`)
`MaxSols` is currently a term which is not a free variable.	(`nonvar/1`)
`Query` is a query.	(`query/1`)
`MaxSols` is an integer.	(`int/1`)
`Results` is a list of `result`s.	(`list/2`)
`NextQuery` is a query.	(`query/1`)

expand_solution/4: PREDICATE

Usage: `expand_solution(Solution,Packages,Fulfilled,Flooded)`

– *Description:* `expand_solution/4` receives a solution `Solution` and gathers more information about the results of the search which lead to that solution:

  * The list of package names returned (`Packages`)
  * which capabilities (either initially requested or internally required by the search process) were found (`Fulfilled`), and
  * which capabilities were needed, but were not satisfied (`Flooded`).

– *The following properties should hold at call time:*

`Solution` is currently a term which is not a free variable.	(`nonvar/1`)
`Solution` is a solution	(`solution/1`)
`Packages` is a list of `atm`s.	(`list/2`)
`Fulfilled` is a list of `search_term`s.	(`list/2`)
`Flooded` is a list of `search_term`s.	(`list/2`)

query_add_search_terms/3: PREDICATE

Usage: `query_add_search_terms(Query,Terms,NewQuery)`

– *Description:* `query_add_search_terms/3` changes query `Query` to query `NewQuery` by adding new search terms `Terms` into it.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable.	(`nonvar/1`)
`Terms` is currently a term which is not a free variable.	(`nonvar/1`)
`Query` is a query.	(`query/1`)
`Terms` is a list of `search_term`s.	(`list/2`)
`NewQuery` is a query.	(`query/1`)

query_remove_search_terms/3: PREDICATE

Usage: `query_remove_search_terms(Query,Terms,NewQuery)`

– *Description:* `query_remove_search_terms/3` changes query `Query` to query `NewQuery` by removing the search terms `Terms` from it.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable.	(`nonvar/1`)
`Terms` is currently a term which is not a free variable.	(`nonvar/1`)
`Query` is a query.	(`query/1`)
`Terms` is a list of `search_term`s.	(`list/2`)
`NewQuery` is a query.	(`query/1`)

query_add_include_packages/3: PREDICATE

Usage: `query_add_include_packages(Query,Packages,NewQuery)`

– *Description:* `query_add_include_packages/3` changes query `Query` to query `NewQuery` by adding new packages `Packages` to appear in a final solution.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable.	(`nonvar/1`)
`Packages` is currently a term which is not a free variable.	(`nonvar/1`)
`Query` is a query.	(`query/1`)
`Packages` is a list of `atm`s.	(`list/2`)
`NewQuery` is a query.	(`query/1`)

query_remove_include_packages/3: PREDICATE

Usage: `query_remove_include_packages(Query,Packages,NewQuery)`

– *Description:* `query_remove_include_packages/3` changes query `Query` to query `NewQuery` by removing `Packages` from the list of packages to appear in a final solution.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable.	(`nonvar/1`)
`Packages` is currently a term which is not a free variable.	(`nonvar/1`)
`Query` is a query.	(`query/1`)
`Packages` is a list of `atm`s.	(`list/2`)
`NewQuery` is a query.	(`query/1`)

query_add_exclude_packages/3: PREDICATE

Usage: `query_add_exclude_packages(Query,Packages,NewQuery)`

– *Description:* `query_add_exclude_packages/3` transforms the query `Query` into the query `NewQuery` by adding the list of packages `Packages` to those which will be excluded from the final solution.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable. (`nonvar/1`)
`Packages` is currently a term which is not a free variable. (`nonvar/1`)
`Query` is a query. (`query/1`)
`Packages` is a list of `atm`s. (`list/2`)
`NewQuery` is a query. (`query/1`)

query_remove_exclude_packages/3: PREDICATE

Usage: `query_remove_exclude_packages(Query,Packages,NewQuery)`

– *Description:* `query_remove_exclude_packages/3` transforms the query `Query` into the query `NewQuery` by removing the list of packages `Packages` from those to be excluded from the final solution.

– *The following properties should hold at call time:*

`Query` is currently a term which is not a free variable. (`nonvar/1`)
`Packages` is currently a term which is not a free variable. (`nonvar/1`)
`Query` is a query. (`query/1`)
`Packages` is a list of `atm`s. (`list/2`)
`NewQuery` is a query. (`query/1`)

heuristic/1: REGTYPE

Usage: `heuristic(H)`

– *Description:* `H` is a valid heuristic, implemented as a ground term. Valid heuristics are:

```
heuristic(all).
heuristic(first(N)) :-
        int(N).
```

solution/1: REGTYPE

Usage: `solution(Sol)`

– *Description:* `Sol` is a solution

query/1: REGTYPE

Usage: `query(Query)`

– *Description:* `Query` is a query.

results/1: REGTYPE

Usage: `results(Results)`

– *Description:* `Results` is a list of solutions.

J Application-Oriented Database Interface

Author(s): Jesús Correas, Manuel Carro, and the CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, `clip@dia.fi.upm.es`, `http://www.clip.dia.fi.upm.es/`.

Version: 0.1#3 (2003/2/18, 12:43:18 CET)

This module implements the predicates which perform queries and modifications on data stored in the database implementation of the AMOS ontology. The primitives defined in this module aim at being implementation-independent: the underlying database technology, schemata, and vendor or query and data language variant should not be reflected at this level. Likewise, the Prolog-level implementation of the data handled by the database interface should not be seen at this level. This is achieved by wrapping each ontology class with a module treating the class instantiations as opaque data structures. The operations provided by these modules are internally used when needed by the database access module.

Unlike the modules related with ontology classes, which have a declarative semantics based on the generation of one state from the previous one, the predicates related to updates in this module are aware of the existence of an (external) state which lives across calls (and, ultimately, across program executions). Therefore, they do not make the distinction of input and output (database) states: changes to the contents of the database are implicit, and states previous to a change are not recoverable on backtracking.

In addition to the operations to consult and store, naturally induced by the ontology, some utility operations, mainly related to accessing the terms needed to perform a search, are included in this module. This is so in order for the matching engine to have a faster access to the information needed to perform searches, and to make the code of the matching engine simpler and clearer.

In order to provide a better integration of the database access primitives with the operational semantics of Prolog, data retrieval operations can both consult and generate answers on backtracking. This makes it easier to use them from within the matching engine as well.

This work has been partially supported by the EU Fifth ESPRIT programme, and it has been developed as part of the AMOS Project (IST-2001-34717).

J.1 Usage and interface (`database_access`)

- **Library usage:**

  `:- use_module(library(database_access)).`

- **Exports:**

  - *Predicates:*

    `resource_description/2`, `resource_needs/2`, `resource_provides/2`, `resource_db_update/2`, `resource_delete/1`, `organization_description/2`, `organization_db_update/2`, `organization_delete/1`, `user_description/2`, `user_db_update/2`, `user_delete/1`, `certification_description/2`, `certification_db_update/2`, `certification_delete/1`, `ditem_description/2`, `ditem_db_update/2`, `ditem_delete/1`.

  - *Regular Types:*

    `db_key/1`.

- **Other modules used:**

  - *Application modules:*

    `asset_access`, `resource_access`, `user_access`, `organization_access`, `certification_access`, `dictionary_access`, `database_impl`.

  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

J.2 Documentation on exports (`database_access`)

db_key/1: REGTYPE

Usage: `db_key(Key)`

- *Description:* `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program.

resource_description/2: PREDICATE

Usage: `resource_description(Desc,Key)`

- *Description:* `Desc` is the complete description of the resource internally identified by `Key`.

- *The following properties should hold at call time:*

  `Desc` is a resource. (`resource/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

resource_needs/2: PREDICATE

Usage 1: `resource_needs(Key,Needs)`

- *Description:* `Needs` is the list of dictionary entries that the package identified by `Key` needs.
- *The following properties should hold at call time:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Needs` is a list of `ditem`s. (`list/2`)

Usage 2: `resource_needs(Res,Needs)`

- *Description:* `Needs` is the list of dictionary entries that the resource `Res` needs.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)

  `Needs` is a list of `ditem`s. (`list/2`)

resource_provides/2: PREDICATE

Usage 1: `resource_provides(Res,Provides)`

- *Description:* `Provides` is the list of dictionary entries that the resource `Res` holds.
- *The following properties should hold at call time:*

  `Res` is a resource. (`resource/1`)

  `Provides` is a list of `ditem`s. (`list/2`)

Usage 2: `resource_provides(Key,Provides)`

- *Description:* `Provides` is the list of dictionary entries that the resource identified by `Key` provides.
- *The following properties should hold at call time:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Provides` is a list of `ditem`s. (`list/2`)

resource_db_update/2: PREDICATE

Usage 1: `resource_db_update(Key,Res)`

- *Description:* The database records are updated in order to make `Res` the new resource description associated to `Key` in the database. `Key` must have been previously generated and associated with a resource. `Res` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `resource_check/2` may be used before updating the database.
- *Call and exit should be* compatible *with:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Res` is a resource. (`resource/1`)
- *The following properties should hold at call time:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Res` is currently ground (it contains no variables). (`ground/1`)
- *The following properties should hold upon exit:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Res` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Res` is a resource. (`resource/1`)

Usage 2: `resource_db_update(Key,Res)`

- *Description:* Resource `Res` is added to the database, and the key associated to the resource is `Key`. If `Res` was not present in the database, a new `Key` is generated for it. `Res` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `resource_check/2` may be used before updating the database.
- *Call and exit should be* compatible *with:*

  `Key` is a free variable. (`var/1`)

  `Res` is a resource. (`resource/1`)
- *The following properties should hold at call time:*

  `Key` is a free variable. (`var/1`)

  `Res` is currently ground (it contains no variables). (`ground/1`)
- *The following properties should hold upon exit:*

  `Res` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Res` is a resource. (`resource/1`)

resource_delete/1: PREDICATE

Usage 1: `resource_delete(ResKey)`

- *Description:* The resource whose associated key is `ResKey` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology.
- *The following properties should hold at call time:*

  `ResKey` is currently ground (it contains no variables). (`ground/1`)

  `ResKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)
- *The following properties should hold upon exit:*

  `ResKey` is currently ground (it contains no variables). (`ground/1`)

Usage 2: `resource_delete(Res)`

- *Description:* The resource represented by `Res` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology. `Res` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `resource_check/2` may be used before updating the database.
- *The following properties should hold at call time:*

  `Res` is currently ground (it contains no variables). (`ground/1`)

  `Res` is a resource. (`resource/1`)
- *The following properties should hold upon exit:*

  `Res` is currently ground (it contains no variables). (`ground/1`)

organization_description/2: PREDICATE

Usage: `organization_description(Desc,Key)`

- *Description:* `Desc` is the complete description of the organization internally identified by `Key`.
- *The following properties should hold at call time:*

  `Desc` is an organization. (`organization/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

organization_db_update/2: PREDICATE

Usage 1: `organization_db_update(Key,Org)`

- *Description:* The database database records are updated in order to make `Org` the new organization description associated to `Key` in the database. `Key` must have been previously generated and associated with an organization. `Org` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `organization_check/2` may be used before updating the database.

- *Call and exit should be* compatible *with:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Org` is an organization. (`organization/1`)

- *The following properties should hold at call time:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Org` is currently ground (it contains no variables). (`ground/1`)

- *The following properties should hold upon exit:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Org` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Org` is an organization. (`organization/1`)

Usage 2: `organization_db_update(Key,Org)`

- *Description:* The organization `Org` is added to the database, and the key associated to the organization description is `Key`. If `Org` was not present in the database, a new `Key` is generated for it. `Org` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `organization_check/2` may be used before updating the database.

- *Call and exit should be* compatible *with:*

  `Key` is a free variable. (`var/1`)

  `Org` is an organization. (`organization/1`)

- *The following properties should hold at call time:*

  `Key` is a free variable. (`var/1`)

  `Org` is currently ground (it contains no variables). (`ground/1`)

- *The following properties should hold upon exit:*

  `Org` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Org` is an organization. (`organization/1`)

organization_delete/1: PREDICATE

Usage 1: `organization_delete(OrgKey)`

- *Description:* The organziation whose associated key is `OrgKey` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology.

- *The following properties should hold at call time:*

  `OrgKey` is currently ground (it contains no variables). (`ground/1`)

  `OrgKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)
- *The following properties should hold upon exit:*

  `OrgKey` is currently ground (it contains no variables). (`ground/1`)

Usage 2: `organization_delete(Org)`

- *Description:* The organization represented by `Org` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology. `Org` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `organization_check/2` may be used before updating the database.
- *The following properties should hold at call time:*

  `Org` is currently ground (it contains no variables). (`ground/1`)

  `Org` is an organization. (`organization/1`)
- *The following properties should hold upon exit:*

  `Org` is currently ground (it contains no variables). (`ground/1`)

user_description/2: PREDICATE

Usage: `user_description(Desc,Key)`

- *Description:* `Desc` is the complete description of the user associated to the identifier `Kay`.
- *The following properties should hold at call time:*

  `Desc` is a user. (`user/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

user_db_update/2: PREDICATE

Usage 1: `user_db_update(Key,User)`

- *Description:* The database database records are updated in order to make `User` the new user description associated to `Key` in the database. `Key` must have been previously generated and associated with an organization. `user` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `user_check/2` may be used before updating the database.
- *Call and exit should be* compatible *with:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `User` is a user. (`user/1`)

– *The following properties should hold at call time:*

`Key` is currently ground (it contains no variables). (`ground/1`)

`User` is currently ground (it contains no variables). (`ground/1`)

– *The following properties should hold upon exit:*

`Key` is currently ground (it contains no variables). (`ground/1`)

`User` is currently ground (it contains no variables). (`ground/1`)

`Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

`User` is a user. (`user/1`)

Usage 2: `user_db_update(Key,User)`

– *Description:* The user `User` is added to the database, and the key associated to the user description is `Key`. If `User` was not present in the database, a new `Key` is generated for it. `user` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `user_check/2` may be used before updating the database.

– *Call and exit should be* compatible *with:*

`Key` is a free variable. (`var/1`)

`User` is a user. (`user/1`)

– *The following properties should hold at call time:*

`Key` is a free variable. (`var/1`)

`User` is currently ground (it contains no variables). (`ground/1`)

– *The following properties should hold upon exit:*

`User` is currently ground (it contains no variables). (`ground/1`)

`Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

`User` is a user. (`user/1`)

user_delete/1: PREDICATE

Usage 1: `user_delete(UserKey)`

– *Description:* The user whose associated key is `UserKey` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology.

– *The following properties should hold at call time:*

`UserKey` is currently ground (it contains no variables). (`ground/1`)

`UserKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

– *The following properties should hold upon exit:*

`UserKey` is currently ground (it contains no variables). (`ground/1`)

Usage 2: `user_delete(User)`

– *Description:* The user represented by `User` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology. `User` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `user_check/2` may be used before updating the database.

– *The following properties should hold at call time:*

`User` is currently ground (it contains no variables). (`ground/1`)

`User` is a user. (`user/1`)

– *The following properties should hold upon exit:*

`User` is currently ground (it contains no variables). (`ground/1`)

certification_description/2: PREDICATE

Usage: `certification_description(Desc,CertKey)`

– *Description:* `Desc` is the complete description of the certification associated to the identifier `CertKey`.

– *The following properties should hold at call time:*

`Desc` is a certification. (`certification/1`)

`CertKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

certification_db_update/2: PREDICATE

Usage 1: `certification_db_update(Key,Cert)`

– *Description:* The database records are updated in order to make `Cert` the new certificate description associated to `Key` in the database. `Key` must have been previously generated and associated with a certificate. `Cert` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `certification_check/2` may be used before updating the database.

– *Call and exit should be* compatible *with:*

`Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

`Cert` is a certification. (`certification/1`)

– *The following properties should hold at call time:*

`Key` is currently ground (it contains no variables). (`ground/1`)

`Cert` is currently ground (it contains no variables). (`ground/1`)

– *The following properties should hold upon exit:*

`Key` is currently ground (it contains no variables). (`ground/1`)

`Cert` is currently ground (it contains no variables). (`ground/1`)

`Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)
`Cert` is a certification. (`certification/1`)

Usage 2: `certification_db_update(Key,Cert)`

- *Description:* The certificate `Cert` is added to the database, and the key associated to the certificate is `Key`. If `Cert` was not present in the database, a new `Key` is generated for it. `Cert` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `certificate_check/2` may be used before updating the database.
- *Call and exit should be* compatible *with:*
  `Key` is a free variable. (`var/1`)
  `Cert` is a certification. (`certification/1`)
- *The following properties should hold at call time:*
  `Key` is a free variable. (`var/1`)
  `Cert` is currently ground (it contains no variables). (`ground/1`)
- *The following properties should hold upon exit:*
  `Cert` is currently ground (it contains no variables). (`ground/1`)
  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)
  `Cert` is a certification. (`certification/1`)

certification_delete/1: PREDICATE

Usage 1: `certification_delete(CertKey)`

- *Description:* The certificate whose associated key is `CertKey` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology.
- *The following properties should hold at call time:*
  `CertKey` is currently ground (it contains no variables). (`ground/1`)
  `CertKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)
- *The following properties should hold upon exit:*
  `CertKey` is currently ground (it contains no variables). (`ground/1`)

Usage 2: `certification_delete(Cert)`

- *Description:* The certificate represented by `Cert` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology. `Cert` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `certification_check/2` may be used before updating the database.

- *The following properties should hold at call time:*

  `Cert` is currently ground (it contains no variables). (`ground/1`)

  `Cert` is a certification. (`certification/1`)

- *The following properties should hold upon exit:*

  `Cert` is currently ground (it contains no variables). (`ground/1`)

ditem_description/2: PREDICATE

Usage: `ditem_description(Desc,Key)`

- *Description:* `Desc` is the description of the dictionary item `Item`.

- *The following properties should hold at call time:*

  `Desc` is one entry of the dictionary. (`ditem/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

ditem_db_update/2: PREDICATE

Usage 1: `ditem_db_update(Key,Item)`

- *Description:* The database records are updated in order to make `Item` the new item description associated to `Key` in the database. `Key` must have been previously generated and associated with an dictionary item. `Item` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `ditem_check/2` may be used before updating the database.

- *Call and exit should be* compatible *with:*

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Item` is one entry of the dictionary. (`ditem/1`)

- *The following properties should hold at call time:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Item` is currently ground (it contains no variables). (`ground/1`)

- *The following properties should hold upon exit:*

  `Key` is currently ground (it contains no variables). (`ground/1`)

  `Item` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Item` is one entry of the dictionary. (`ditem/1`)

Usage 2: `ditem_db_update(Key,Item)`

- *Description:* The dictionary item `Item` is added to the database, and the key associated to the item description is `Key`. If `Item` was not present in the database, a new `Key` is generated for it. `Item` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `ditem_check/2` may be used before updating the database.
- *Call and exit should be* compatible *with:*

  `Key` is a free variable. (`var/1`)

  `Item` is one entry of the dictionary. (`ditem/1`)
- *The following properties should hold at call time:*

  `Key` is a free variable. (`var/1`)

  `Item` is currently ground (it contains no variables). (`ground/1`)
- *The following properties should hold upon exit:*

  `Item` is currently ground (it contains no variables). (`ground/1`)

  `Key` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

  `Item` is one entry of the dictionary. (`ditem/1`)

ditem_delete/1: PREDICATE

Usage 1: `ditem_delete(ItemKey)`

- *Description:* The dictionary item whose associated key is `ItemKey` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology.
- *The following properties should hold at call time:*

  `ItemKey` is currently a term which is not a free variable. (`nonvar/1`)

  `ItemKey` is a (Prolog) representation of an unique indexing key, used by the database. It is to be changed according the the database implementation, and it should not be changed or manipulated by the Prolog program. (`db_key/1`)

Usage 2: `ditem_delete(Item)`

- *Description:* The dictionary entry represented by `Item` is removed from the database, along with all the information which would make the database incoherent with respect to the constraints stated in the ontology. `Item` must not contain free variables, and it must abide by the integrity constraints of the ontology. In order to ensure that, `ditem_check/2` may be used before updating the database.
- *The following properties should hold at call time:*

  `Item` is currently a term which is not a free variable. (`nonvar/1`)

  `Item` is one entry of the dictionary. (`ditem/1`)

J.3 Version/Change Log (`database_access`)

Version 0.1#3 (2003/2/18, 12:43:18 CET) preliminary database implementation and minor documentation changes (Jesus Correas Fernandez)

Version 0.1#2 (2003/2/17, 16:1:32 CET) Documentation and minor changes on module interface. (Jesus Correas Fernandez)

Version 0.1#1 (2003/2/14, 15:56:45 CET) Initial version, with all interfaces included. (Jesus Correas Fernandez)

References

[ACFLGP01] J. C. Arpírez, O. Corcho, M. Fernández-López, and A. Gómez-Pérez. Webode: a Scalable Workbench for Ontological Engineering. In *Proceedings of the International Conference on Knowledge Capture*, pages 6–13. ACM Press, 2001.

[Daf02] Carlo Daffara. An ontology for open source code. Technical report, Conecta s.r.l., 2002. Deliverable D2 of the AMOS Project.

[Far95] Adam Farquhar. Ontolingua to Prolog Syntax Translation. Available at `http://ksl-web.stanford.edu/people/axf/ol-to-prolog.txt`, April 1995.

[KW02] M. M. Kokar and J. Wang. An Example of Using Ontologies and Symbolic Information in Automatic Target Recognition. In *Sensor Fusion: Architectures, Algorithms and Applications VI*, volume 4731 of *SPIE*, pages 40–50, 2002.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and research directions. *IEEE Transactions on Software Engineering*, 1995.

[SPRS02] D. Sleeman, S. Potter, D. Robertson, and M. Schorlemmer. Ontology Extraction for Distributed Environments. In *Workshop on Knowledge Transformations for the Semantic Web (affiliated to ECAI-02)*, July 2002.

[Tec] AKT Technologies. Edinburgh Mission Statement. Available at `http://www.aktors.org/publications/technologies/extrakt/`.

Index