SWI-Prolog ODBC Interface

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: jan@swi-prolog.org

February 2, 2009

Abstract

This document describes the SWI-Prolog interface to ODBC, the Microsoft standard for *Open DataBase Connectivity*. These days there are ODBC managers from multiple vendors for many platforms as well as drivers for most databases, making it an attractive target for a Prolog database connection.

The database interface is envisioned to consist of two layers. The first layer is an encapsulation of the core functionality of ODBC. This layer makes it possible to run SQL queries. The second layer exploits the relation between Prolog predicates and database tables, providing —a somewhat limited— natural Prolog view on the data. The current interface only covers the first layer.

Contents

1 Introduction

The value of RDMS for Prolog is often over-estimated, as Prolog itself can manage substantial amounts of data. Nevertheless a Prolog/RDMS interface provides advantages if data is already provided in an RDMS, data must be shared with other applications, there are strong persistency requirements or there is too much data to fit in memory.

The popularity of ODBC makes it possible to design a single foreign-language module that provides RDMS access for a wide variety of databases on a wide variety of platforms. The SWI-Prolog RDMS interface is closely modeled after the ODBC API. This API is rather low-level, but defaults and dynamic typing provided by Prolog give the user quite simple access to RDMS, while the interface provides the best possible performance given the RDMS independency constraint.

The Prolog community knows about various high-level connections between RDMS and Prolog. We envision these layered on top of the ODBC connection described here.

2 The ODBC layer

2.1 Connection management

The ODBC interface deals with a single ODBC environment with multiple simultaneous connections. The predicates in this section deal with connection management.

odbc_connect(+*DSN, -Connection, +Options*)

Create a new ODBC connection to data-source *DSN* and return a handle to this connection in *Connection*. The connection handle is either an opaque structure or an atom of the `alias` option is used. In addition to the options below, options applicable to `odbc_set_connection/2` may be provided.

user(*User*)

Define the user-name for the connection. This option must be present if the database uses authorization.

password(*Password*)

Provide a password for the connection. Normally used in combination with `user`(*User*).

alias(*AliasName*)

Use *AliasName* as *Connection* identifier, making the connection available as a global resource. A good choice is to use the *DSN* as alias.

open(*OpenMode*)

If *OpenMode* is `once` (default if an `alias` is provided), a second call to open the same *DSN* simply returns the existing connection. If `multiple` (default if there is no alias name), a second connection to the same data-source is opened.

mars(+*Bool*)

If `true`, use Microsoft SQL server 2005 *mars* mode. This is support for multiple concurrent statements on a connection without requiring the dynamic cursor (which incurs an astounding 20-50x slowdown of query execution!!). MARS is a new feature in SQL2k5 apparently, and only works if you use the native driver. For the non-native driver, specifying that it is enabled will have absolutely no effect.

The following example connects to the WordNet[1] [**?**] database, using the connection alias `wordnet` and opening the connection only once:

```
open_wordnet :-
        odbc_connect('WordNet', _,
                     [ user(jan),
                       password(xxx),
                       alias(wordnet),
                       open(once)
                     ]).
```

odbc_disconnect(+*Connection*)
Close the given *Connection*. This destroys the connection alias or, if there is no alias, makes further use of the *Connection* handle illegal.

odbc_current_connection(*?Connection, ?DSN*)
Enumerate the existing ODBC connections.

odbc_set_connection(+*Connection, +Option*)
Set options on an existing connection. All options defined here may also be specified with `odbc_connect/2` in the option-list. Defined options are:

access_mode(*Mode*)
If `read`, tell the driver we only access the database in read mode. If `update` (default), tell the driver we may execute update commands.

auto_commit(*bool*)
If `true` (default), each update statement is committed immediately. If `false`, an update statement starts a transaction that can be committed or rolled-back. See section **??** for details on transaction management.

cursor_type(*CursorType*)
I haven't found a good description of what this does, but setting it to `dynamic` makes it possible to have multiple active statements on the same connection with Microsoft SQL server. Other values are `static`, `forwards_only` and `keyset_driven`.

silent(*Bool*)
If `true` (default `false`), statements returning `SQL_SUCCESS_WITH_INFO` succeed without printing the info. See also section **??**.

null(*NullSpecifier*)
Defines how the SQL constant NULL is represented. Without specification, the default is the atom `$null$`. *NullSpecifier* is an arbitrary Prolog term, though the implementation is optimised for using an unbound variable, atom and functor with one unbound variable. The representation `null(_)` is a commonly used alternative.

The specified default holds for all statements executed on this connection. Changing the connection default does not affect already prepared or running statements. The null-value can also be specified at the statement level. See the option list of `odbc_query/4`.

[1] An SQL version of WordNet is available from `http://wordnet2sql.infocity.cjb.net/`

wide_column_threshold(+*Length*)

If the width of a column exceeds *Length*, use the API SQLGetData() to get the value incrementally rather than using a (large) buffer allocated with the statement. The default is to use this alternate interface for columns larger than 1024 bytes. There are two cases for using this option. In time critical applications with wide columns it may provide better performance at the cost of a higher memory usage and to work around bugs in SQLGetData(). The latter applies to Microsoft SQL Server fetching the definition of a view.

odbc_get_connection(+*Connection, ?Property*)

Query for properties of the connection. *Property* is a term of the format *Name*(*Value*). If *Property* is unbound all defined properties are enumerated on backtracking. Currently the following properties are defined.

database_name(*Atom*)

Name of the database associated to the connection.

dbms_name(*Name*)

Name of the database engine. This constant can be used to identify the engine.

dbms_version(*Atom*)

Version identifier from the database engine.

driver_name(*Name*)

ODBC Dynamic Link Library providing the interface between ODBC and the database.

driver_odbc_version(*Atom*)

ODBC version supported by the driver.

driver_version(*Atom*)

The drivers version identifier.

active_statements(*Integer*)

Maximum number of statements that can be active at the same time on this connection. Returns 0 (zero) if this is unlimited.[2]

odbc_data_source(*?DSN, ?Description*)

Query the defined data sources. It is not required to have any open connections before calling this predicate. *DSN* is the name of the data source as required by `odbc_connect/3`. *Description* is the name of the driver. The driver name may be used to tailor the SQL statements used on the database. Unfortunately this name depends on the local installing details and is therefore not universally useful.

2.2 Running SQL queries

ODBC distinguishes between direct execution of literal SQL strings and parameterized execution of SQL strings. The first is a simple practical solution for infrequent calls (such as creating a table), while parameterized execution allows the driver and database to precompile the query and store the optimized code, making it suitable for time-critical operations. In addition, it allows for passing parameters without going through SQL-syntax and thus avoiding the need for quoting.

[2]Microsoft SQL server can have multiple active statements after setting the option `cursor_type` to `dynamic`. See `odbc_set_connection/2`.

2.2.1 One-time invocation

odbc_query(+*Connection, +SQL, -RowOrAffected*)
Same as `odbc_query/4` using `[]` for *Options*.

odbc_query(+*Connection, +SQL, -RowOrAffected, +Options*)
Fire an SQL query on the database represented by *Connection*. *SQL* is any valid SQL statement. SQL statements can be specified as a plain atom, string or a term of the format *Format-Arguments*, which is converted using `format/2`.

If the statement is a `SELECT` statement the result-set is returned in *RowOrAffected*. By default rows are returned one-by-one on backtracking as terms of the functor `row/`*Arity*, where *Arity* denotes the number of columns in the result-set. The library pre-fetches the next value to be able to close the statement and return deterministic success when returning the last row of the result-set. Using the option `findall/2` (see below) the result-set is returned as a list of user-specified terms. For other statements this argument returns `affected`(*Rows*), where *Rows* represents the number of rows affected by the statement. If you are not interested in the number of affected rows `odbc_query/2` provides a simple interface for sending SQL-statements.

Below is a small example using the connection created from `odbc_connect/3`. Please note that the SQL-statement does not end in the '`;`' character.

```
lemma(Lemma) :-
        odbc_query(wordnet,
                   'SELECT (lemma) FROM word',
                   row(Lemma).
```

The following example adds a name to a table with parent-relations, returning the number of rows affected by the statement.

```
insert_child(Child, Mother, Father, Affected) :-
        odbc_query(parents,
                   'INSERT INTO parents (name,mother,father) \
                      VALUES ("mary", "christine", "bob")',
                   affected(Affected)).
```

Options defines the following options.

types(*ListOfTypes*)
Determine the Prolog type used to report the column-values. When omitted, default conversion as described in section **??** is implied. A column may specify `default` to use default conversion for that column. The length of the type-list must match the number of columns in the result-set.

For example, in the table `word` the first column is defined with the SQL type `DECIMAL(6)`. Using this SQL-type, "001" is distinct from "1", but using Prolog integers is a valid representation for Wordnet `wordno` identifiers. The following query extracts rows using Prolog integers:

```
?- odbc_query(wordnet,
              'select * from word', X,
              [ types([integer,default])
              ]).

X = row(1, entity) ;
X = row(2, thing) ;
...
```

See also section **??** for notes on type-conversion.

null(*NullSpecifier*)

Specify SQL NULL representation. See `odbc_set_connection/2` for details.

source(*Bool*)

If `true` (default `false`), include the source-column with each result-value. With this option, each result in the `row/`*N*-term is of the format below. *TableName* or *ColumnName* may be the empty atom if the information is not available.[3]

`column`(*TableName, ColumnName, Value*)

findall(*Template, row(Column, ...)*)

Instead of returning rows on backtracking this option makes `odbc_query/3` return all rows in a list and close the statement. The option is named after the Prolog `findall/3` predicate, as the it makes `odbc_query/3` behave as the commonly used `findall/3` construct below.

```
lemmas(Lemmas) :-
        findall(Lemma,
                odbc_query(wordnet,
                           'select (lemma) from word',
                           row(Lemma)),
                Lemmas).
```

Using the `findall/2` option the above can be implemented as below. The number of argument of the `row` term must match the number of columns in the result-set.

```
lemmas(Lemmas) :-
        odbc_query(wordnet,
                   'select (lemma) from word',
                   Lemmas,
                   [ findall(Lemma, row(Lemma))
                   ]).
```

The current implementation is incomplete. It does not allow arguments of `row(...)` *to be instantiated. Plain instantiation can always be avoided using a proper SELECT statement. Potentially useful however would be the translation of compound terms, especially to translate date/time/timestamp structures to a format for use by the application.*

[3] This is one possible interface to this information. In many cases it is more efficient and convenient to provide this information separately as it is the same for each result-row.

wide_column_threshold(+*Length*)
Specify threshold column width for using SQLGetData(). See `odbc_set_connection/2` for details.

odbc_query(+*Connection, +SQL*)
As `odbc_query/3`, but used for SQL-statements that should not return result-rows (i.e. all statements except for `SELECT`). The predicate prints a diagnostic message if the query returns a result.

2.2.2 Parameterised queries

ODBC provides for 'parameterized queries'. These are SQL queries with a `?`-sign at places where parameters appear. The ODBC interface and database driver may use this to precompile the SQL-statement, giving better performance on repeated queries. This is exactly what we want if we associate Prolog predicates to database tables. This interface is defined by the following predicates:

odbc_prepare(+*Connection, +SQL, +Parameters, -Statement*)
As `odbc_prepare/5` using `[]` for *Options*.

odbc_prepare(+*Connection, +SQL, +Parameters, -Statement, +Options*)
Create a statement from the given *SQL* (which may be a format specification as described with `odbc_query/3`) statement that normally has one or more parameter-indicators (`?`) and unify *Statement* with a handle to the created statement. *Parameters* is a list of descriptions, one for each parameter. Each parameter description is one of the following:

default
Uses the ODBC function SQLDescribeParam() to obtain information about the parameter and apply default rules. See section **??** for details. If the interface fails to return a type or the type is unknown to the ODBC interface a message is printed and the interface handles the type as text, which implies the user must supply an atom. The message can be suppressed using the `silent`(*true*) option of `odbc_set_connection/2`. An alternative mapping can be selected using the > option of this predicate described below.

SqlType(*Specifier, ...*)
Declare the parameter to be of type *SqlType* with the given specifiers. Specifiers are required for `char`, `varchar`, etc. to specify the field-width. When calling `odbc_execute/[2-3]`, the user must supply the parameter values in the default Prolog type for this SQL type. See section **??** for details.

PrologType > *SqlType*
As above, but supply values of the given *PrologType*, using the type-transformation defined by the database driver. For example, if the parameter is specified as

```
atom > date
```

The use must supply an atom of the format `YYYY-MM-DD` rather than a term `date`(*Year,Month,Day*). This construct enhances flexibility and allows for passing values that have no proper representation in Prolog.

Options defines a list of options for executing the statement. See `odbc_query/4` for details. In addition, the following option is provided:

fetch(*FetchType*)

Determine the *FetchType*, which is one of `auto` (default) to extract the result-set on backtracking or `fetch` to prepare the result-set to be fetched using `odbc_fetch/3`.

odbc_execute(+*Statement, +ParameterValues, -RowOrAffected*)

Execute a statement prepared with `odbc_prepare/4` with the given *ParameterValues* and return the rows or number of affected rows as `odbc_query/4`. This predicate may return type_error exceptions if the provided parameter values cannot be converted to the declared types.

ODBC doesn't appear to allow for multiple cursors on the same result-set.[4] This would imply there can only be one active `odbc_execute/3` (i.e. with a choice-point) on a prepared statement. Suppose we have a table `age (name char(25), age integer)` bound to the predicate `age/2` we cannot write the code below without special precautions. The ODBC interface therefore creates a clone of a statement if it discovers the statement is being executed, which is discarded after the statement is finished.[5]

```
same_age(X, Y) :-
        age(X, AgeX),
        age(Y, AgeY),
        AgeX = AgeY.
```

odbc_execute(+*Statement, +ParameterValues*)

Like `odbc_query/2`, this predicate is meant to execute simple SQL statements without interest in the result.

odbc_free_statement(+*Statement*)

Destroy a statement prepared with `odbc_prepare/4`. If the statement is currently executing (i.e. `odbc_execute/3` left a choice-point), the destruction is delayed until the execution terminates.

2.2.3 Fetching rows explicitely

Normally SQL queries return a result-set that is enumerated on backtracking. Using this approach a result-set is similar to a predicate holding facts. There are some cases where fetching the rows one-by-one, much like `read/1` reads terms from a file is more appropriate and there are cases where only part of the result-set is to be fetched. These cases can be dealt with using `odbc_fetch/3`, which provides an interface to SQLFetchScroll().

As a general rule of thumb, stay away from these functions if you do not really need them. Experiment before deciding on the strategy and often you'll discover the simply backtracking approach is much easier to deal with and about as fast.

odbc_fetch(+*Statement, -Row, +Option*)

Fetch a row from the result-set of *Statement*. *Statement* must be created with `odbc_prepare/5` using the option `fetch`(*fetch*) and be executed using `odbc_execute/2`.

[4] Is this right?

[5] The code is prepared to maintain a cache of statements. Practice should tell us whether it is worthwhile activating this.

Row is unified to the fetched row or the atom `end_of_file`[6] after the end of the data is reached. Calling `odbc_fetch/2` after all data is retrieved causes a permission-error exception. *Option* is one of:

next
: Fetch the next row.

prior
: Fetch the result-set going backwards.

first
: Fetch the first row.

last
: Fetch the last row.

absolute(*Offset***)**
: Fetch absolute numbered row. Rows count from one.

relative(*Offset***)**
: Fetch relative to the current row. `relative(`*1*`)` is the same as `next`, except that the first row extracted is row 2.

bookmark(*Offset***)**
: Reserved. Bookmarks are not yet supported in this interface.

In many cases, depending on the driver and RDBMS, the cursor-type must be changed using `odbc_set_connection/2` for anything different from `next` to work.

Here is example code each time skipping a row from a table 'test' holding a single column of integers that represent the row-number. This test was executed using unixODBC and MySQL on SuSE Linux.

```
fetch(Options) :-
        odbc_set_connection(test, cursor_type(static)),
        odbc_prepare(test,
                     'select (testval) from test',
                     [],
                     Statement,
                     [ fetch(fetch)
                     ]),
        odbc_execute(Statement, []),
        fetch(Statement, Options).

fetch(Statement, Options) :-
        odbc_fetch(Statement, Row, Options),
        (   Row == end_of_file
        ->  true
        ;   writeln(Row),
            fetch(Statement, Options)
        ).
```

[6] This atom was selected to emphasise the similarity to read.

odbc_close_statement(*C***)**
loses the given statement (without freeing it). This must be used if not the whole result-set is retrieved using `odbc_fetch/3`.

2.3 Transaction management

ODBC can run in two modi. By default, all update actions are immediately committed on the server. Using `odbc_set_connection/2` this behaviour can be switched off, after which each SQL statement that can be inside a transaction implicitly starts a new transaction. This transaction can be ended using `odbc_end_transaction/2`.

odbc_end_transaction(*+Connection, +Action***)**
End the currently open transaction if there is one. Using *Action* `commit` pending updates are made permanent, using `rollback` they are discarded.

The ODBC documentation has many comments on transaction management and its interaction with database cursors.

2.4 Accessing the database dictionary

With this interface we do not envision the use of Prolog as a database manager. Nevertheless, elementary access to the structure of a database is required, for example to validate a database satisfies the assumptions made by the application.

odbc_current_table(*+Connection, -Table***)**
Return on backtracking the names of all tables in the database identified by the connection.

odbc_current_table(*+Connection, ?Table, ?Facet***)**
Enumerate properties of the tables. Defines facets are:

qualifier(*Qualifier***)**

owner(*Owner***)**

comment(*Comment***)**
These facets are defined by SQLTables()

arity(*Arity***)**
This facet returns the number of columns in a table.

odbc_table_column(*+Connection, ?Table, ?Column***)**
On backtracking, enumerate all columns in all tables.

odbc_table_column(*+Connection, ?Table, ?Column, ?Facet***)**
Provides access to the properties of the table as defined by the ODBC call SQLColumns(). Defined facets are:

table_qualifier(*Qualifier***)**

table_owner(*Owner*)

table_name(*Table*)
See `odbc_current_table/3`.

data_type(*DataType*)

type_name(*TypeName*)

precision(*Precision*)

length(*Length*)

scale(*Scale*)

radix(*Radix*)

nullable(*Nullable*)

remarks(*Remarks*)
These facets are defined by SQLColumns()

type(*Type*)
More prolog-friendly representation of the type properties. See section **??**.

odbc_type(+*Connection, ?TypeSpec, ?Facet*)
Query the types supported by the data source. *TypeSpec* is either an integer type-id, the name of an ODBC SQL type or the constant `all_types` to enumerate all known types. This predicate calls SQLGetTypeInfo() and its facet names are derived from the specification of this ODBC function:

name(*Name*)
Name used by the data-source. Use this in CREATE statements

data_type(*DataType*)
Numeric identifier of the type

precision(*Precision*)
When available, maximum precision of the type.

literal_prefix(*Prefix*)
When available, prefix for literal representation.

literal_suffix(*Suffix*)
When available, suffix for literal representation.

create_params(*CreateParams*)
When available, arguments needed to create the type.

nullable(*Bool*)
Whether the type can be `NULL`. May be `unknown`

case_sensitive(*Bool*)
Whether values for this type are case-sensitive.

searchable(*Searchable*)
Whether the type can be searched. Values are `false`, `true`, `like_only` or `all_except_like`.

unsigned(*Bool*)
When available, whether the value is signed. Please note that SWI-Prolog does not provide unsigned integral values.

money(*Bool*)
Whether the type represents money.

auto_increment(*Bool*)
When available, whether the type can be auto-incremented.

local_name(*LocalName*)
Name of the type in local language.

minimum_scale(*MinScale*)
Minimum scale of the type.

maximum_scale(*MaxScale*)
Maximum scale of the type.

2.5 Getting more information

odbc_statistics(*?Key*)
Get statistical data on the ODBC interface. Currently defined keys are:

statements(*Created, Freed*)
Number of SQL statements that have been *Created* and *Freed* over all connections. Statements executed with `odbc_query/[2-3]` increment *Created* as the query is created and *Freed* if the query is terminated due to deterministic success, failure, cut or exception. Statements created with `odbc_prepare/[4-5]` are freed by `odbc_free_statement/1` or due to a fatal error with the statement.

odbc_debug(+*Level*)
Set the verbosity-level to *Level*. Default is 0. Higher levels make the system print debugging messages.

2.6 Representing SQL data in Prolog

Databases have a poorly standardized but rich set of datatypes. Some have natural Prolog counterparts, some not. A complete mapping requires us to define Prolog data-types for SQL types that have no standardized Prolog counterpart (such as timestamp), the definition of a default mapping and the possibility to define an alternative mapping for a specific column. For example, many variations of the SQL `DECIMAL` type cannot be mapped to a Prolog integer. Nevertheless, mapping to an integer may be the proper choice for a specific application.

The Prolog/ODBC interface defines the following Prolog result types with the indicated default transformation. Different result-types can be requested using the `types`(*TypeList*) option for the `odbc_query/4` and `odbc_prepare/5` interfaces.

atom
: Used as default for the SQL types `char`, `varchar`, `longvarchar`, `binary`, `varbinary`, `longvarbinary`, `decimal` and `numeric`. Can be used for all types.

string
: SWI-Prolog extended type string. Use the type for special cases where garbage atoms must be avoided. Can be used for all types.

codes
: List of character codes. Use this type if the argument must be analysed or compatibility with Prolog systems that cannot handle infinite-length atoms is desired. Can be used for all types.

integer
: Used as default for the SQL types `bit`, `tinyint`, `smallint` and `integer`. Please note that SWI-Prolog integers are signed 32-bit values, where SQL allows for unsigned values as well. Can be used for the integral, and `decimal` types as well as the types `date` and `timestamp`, which are represented as POSIX time-stamps (seconds after Jan 1, 1970).

double
: Used as default for the SQL types `real`, `float` and `double`. Can be used for the integral, float and `decimal` types as well as the types `date` and `timestamp`, which are represented as POSIX time-stamps (seconds after Jan 1, 1970). Representing time this way is compatible to SWI-Prologs time-stamp handling.

date
: A Prolog term of the form `date`(*Year,Month,Day*) used as default for the SQL type `date`.

time
: A Prolog term of the form `time`(*Hour,Minute,Second*) used as default for the SQL type `time`.

timestamp
: A Prolog term of the form `timestamp`(*Year,Month,Day,Hour,Minute,Second,Fraction*) used as default for the SQL type `timestamp`.

2.7 Errors and warnings

ODBC operations return success, error or 'success with info'. This section explains how results from the ODBC layer are reported to Prolog.

2.7.1 ODBC messages: 'Success with info'

If an ODBC operation returns 'with info', the info is extracted from the interface and handled to the Prolog message dispatcher `print_message/2`. The level of the message is `informational` and the term is of the form:

odbc(*State, Native, Message*)
: Here, *State* is the SQL-state as defined in the ODBC API, *Native* is the (integer) error code of the underlying data source and *Message* is a human readable explanation of the message.

2.7.2 ODBC errors

If an ODBC operation signals an error, it throws the exception `error(odbc(`*State, Native, Message*`), _)`. The arguments of the `odbc/3` term are explained in section **??**.

In addition, the Prolog layer performs the normal tests for proper arguments and state, signaling the conventional instantiation, type, domain and resource exceptions.

2.8 ODBC implementations

There is a wealth on ODBC implementations that are completely or almost compatible to this interface. In addition, a number of databases are delivered with an ODBC compatible interface. This implies you get the portability benefits of ODBC without paying the configuration and performance price. Currently this interface is, according to the PHP documentation on this subject, provided by Adabas D, IBM DB2, Solid, and Sybase SQL Anywhere.

2.8.1 Using unixODBC

The SWI-Prolog ODBC interface was developed using unixODBC and MySQL on SuSE Linux.

2.8.2 Using Microsoft ODBC

On MS-Windows, the ODBC interface is a standard package, linked against `odbc32.lib`.

2.9 Remaining issues

The following issues are identified and waiting for concrete problems and suggestions.

Transaction management This certainly requires a high-level interface. Possibly in combination with `call_cleanup/3`, providing automatic rollback on failure or exception and commit on success.

High-level interface Attaching tables to predicates, partial *DataLog* implementation, etc.

3 Installation

3.1 Unix systems

Installation on Unix system uses the commonly found *configure*, *make* and *make install* sequence. SWI-Prolog should be installed before building this package. If SWI-Prolog is not installed as `pl`, the environment variable `PL` must be set to the name of the SWI-Prolog executable. Installation is now accomplished using:

```
% ./configure
% make
% make install
```

This installs the foreign libraries in `$PLBASE/lib/$PLARCH` and the Prolog library files in `$PLBASE/library`, where `$PLBASE` refers to the SWI-Prolog 'home-directory'.

4 Acknowledgments

The SWI-Prolog ODBC interface started from a partial interface by Stefano De Giorgi. Mike Elston suggested programmable null-representation with many other suggestions while doing the first field-tests with this package.