The Athomaris Library
A Framework for Business Processes, Database Integration, and Web Interfaces

Thomas Schöbel-Theuer

Version 0.5

Technical features:

- Automated **business processes**: **orchestration level**, and execution by a very generic **state transition machine**, with generic interfaces to almost arbitrary service providers.
- Provides ***temporal* databases**: the **history** of all changes may be recorded, even deletions. Temporal queries possible, in addition to non-temporal views of the data.
- **Versioned schema management**: when creating new versions of `$SCHEMA`, SQL statements like `alter table` are generated. Live data is left intact whenever possible.
- **Access management**: fine-grained access control based on user profiles. The user interface obeys these rules by presenting only those fields and buttons for which access is currently allowed.
- **Distributed access** to heterogeneous databases. Logical integration of multiple data sources into a single **"virtual database"** with **cross-join capabilities** and **cross-referential integrity** are in preparation.
- **Scalable**: transparent support for master/slave replication setups in large enterprises. **Vertical splitting** via views. Transparent **horizontal splitting** is in preparation.
- **Automatic consistency updates**: tables on different (distributed) databases may be kept always in-sync with each other, either symmetrically or asymmetrically in multiple modes.
- Generic **default user interface**: once `$SCHEMA` has been created, the database can be accessed instantly via a customizable web-based interface (without programming).
- Internal **Model-View-Controller** (MVC) architecture: generic **template management** with internationalization and user-profile support. Customizable and extensible.
- **Security**: generated SQL code and templates are automatically escaped to prevent attacks like SQL injection, XSS, etc.
- Further aspects: see FAQ in the appendix (last page).

Contents

1 Basic Setup

Requirements: you need Linux, Apache with `mod_php5` and some of the supported databases (currently only MySQL, but a Sybase driver and other drivers are planned). If you want to use the business process execution engine, you also need to install and enable the `pcntl` extension module of PHP.

Copy the sources of the Athomaris PHP Library to the document root of your webserver. In the following, we will assume that this is `/www`. If you use another place, you have to adapt the following examples accordingly. So it looks like

- `cd /www; sudo tar xzpf athomaris.tar.gz`

This creates a directory `/www/athomaris/`. It is imporant to do this as `root`, because there are directories owned by different users[1]. Afterwards, configure Apache the following way:

```
<Directory "/www/athomaris">
    Options ExecCGI FollowSymLinks
    AllowOverride AuthConfig
    AuthType Basic
    AuthName "The Athomaris PHP Library"
    AuthBasicProvider file anon
    AuthUserFile /dev/null
    Require valid-user
    AuthBasicAuthoritative Off
    Anonymous_NoUserID off
    Anonymous *
    Order allow,deny
    Allow from all
</Directory>
Alias /demo_basic "/www/athomaris/demo_basic"
Alias /demo_advanced "/www/athomaris/demo_advanced"
Alias /demo_business "/www/athomaris/demo_business"
```

Make sure the `authn_anon` module of Apache is loaded. Many Linux distributions have it disabled by default. Typically, you need to add it to `/etc/sysconfig/apache2` or to some file in `/etc/apache2/sysconfig.d/` or to another place within `/etc/apache2/` (distribution specific). Afterwards don't forget to restart Apache!

Important: if you communicate over **untrusted networks**, is is *highly recommended* to use `https` instead of `http`. Otherwise an attacker will be able to watch your passwords from the basic HTTP authentification. You will need an `SSL` certificate for `https`. Creating or obtaining certificates and configuring Apache for `https` is beyond the scope of this paper. Please consult the Apache documentation and various web resources for that.

Further potential pitfalls: if you want to process large data fields such as TEXT or BLOBs, you probably have to increase the *memory limits* of PHP. Some distributions use rather small defaults.

1.1 Demo Projects

Provided you have a local MySQL installation running, you can immediately use the demo projects `/www/athomaris/demo_basic/`, `/www/athomaris/demo_advanced/`, and `/www/athomaris/demo_business/`.

[1] If it went wrong, or if you got the sources via `svn`, you can the fix the permissions as follows:

- `cd /www/athomaris; chmod 755 */compiled; sudo chown -R wwwrun */compiled/`

Inspect the files in the demo project. There are not many, and they are rather small and simple. For example, `config.php` in each of the demo directories describes the connections to database servers and their drivers. The others mostly include generic code from the `/www/athomaris/common/` subdirectory.

The database schema in `/www/athomaris/demo_basic/schema/schema01.php` is the most central place you will have to deal with. Its content should be rather self-explanatory for PHP programmers. Details on it will be provided later.

Use a web browser such as Firefox or Konqueror to open the following URL:

- `http://localhost/demo_basic/create_schema.php`

If your Apache setup is correct, your browser will ask you for a username and a password (using the basic authentification mechanism of the HTTP protocol). Type in `root` and the appropriate root password for your MySQL server. Note that you *must* have already set a root password for MySQL (which is not the same as the root password of the operating system), and you normally really *need*[2] root access in order to be able to create a new database[3]. If you have trouble with that, consult the MySQL and the Apache documentation, inspect the logfiles, and probably turn on debugging of some of the components.

After successfully gaining root access, you will see a *preview* screen with MySQL commands. Look at them. If you want to actually execute these commands, click on the button named `demo_advanced`. If all is ok, you will be notified that the database has been created.

If you like, you may check the database by hand, for example:

- `mysql -p -u root demo_basic`
- `show tables;`
- `describe foos;`

You can see that the schema definition from `schema/schema01.php` has been applied pretty straightforward. However, some additional columns have been *automatically* created. Some of them have to do with *temporal databases* (see section 2.1.3).

Now let us change the schema, by adding a new column. Copy the file `schema/schema01.php` to `schema/schema02.php` without altering the old version s`chema01.php`, and afterwards edit the new version `schema02.php` in the following way:

Copy the definition of `foo_name` including its sub-structure, and rename it to `foo_somethingelse`. You may change the `TYPE` field to a different type, such as `varchar(50)`, or change the `DEFAULT` value to something you like. A list of available types will be provided later. Note that you must always provide a `DEFAULT` value when you want to create a *new* column in the database. You can also change the `DEFAULT` of an existing column, or even remove a column (but please don't remove columns mentioned in UNIQUE keys - they are vital).

After modification of `schema02.php`, just open the URL for `create_schema.php` again. Now you will see different SQL statements, in particular `alter table` or `drop column` statements, depending on what you have changed between `schema01.php` and `schema02.php`. When executed, these statements will update the database to the new version of the schema.

> What's the "big clue" with that? When using advanced features such as `profiles` (see `demo_advanced`) or when implicitly adding further tables (e.g. for the business process engine, see section 4), these tables will be automatically *kept consistent* with the global schema $\Longrightarrow$ aka **schema management**. In future Athomaris releases, we want to provide web interfaces for schema management, then you no longer will need to deal with PHP variables.

In order to use the database, open the following URL:

- `http://localhost/demo_basic/`

[2] Exception: you are a database expert who knows how to grant appropriate privileges to other users.

[3] Some Linux distros deliver MySQL without any root password. In such a case Athomaris will not work, because passwords are mandatory. You may use the following command for setting an initial password:

- `mysqladmin -u root password ‘‘secret’’`

You can enter new tuples into the database, browse the data in various orders and selections, update and delete the data as you like. It should be rather self-explanatory what you can do. Just try it!

If you like, you may change the schema again using the name `schema03.php`, after you already have populated your tables with data.

If you want to customize the user interface, look at the file `/www/athomaris/common/tpl/generic/generic.tpl` and copy some of its template definitions over to `/www/athomaris/demo_basic/lang/generic/generic.tpl`. There you may modify the HTML code (details on the template syntax, macro substitutions, hooks are in section 3.3). After having done that, you must restart the template compiler via the following URL:

- `http://localhost/demo_basic/translate_tpl.php`

When run successfully, it translates all the templates to PHP code which you can find in `compiled/generic_generic.php`. This code does not aim to be readable, but is very fast on execution.

1.2 New Projects

The easiest way to start a new project is by copying one of the demo projects (or any other project) to a new subdirectory. For example

- `cd /www/athomaris/; sudo cp -a demo_business myproject`

It is important to use the `-a` flag as `root`, in order to keep some symlinks intact and to ensure that the subdirectory `compiled/` remains owned by the Apache user and is writeable by it.

Afterwards, you should configure Apache such that your new directory is accessible via HTTP over some URL. For example, add the following to `httpd.conf`:

- `Alias /myproject "/www/athomaris/myproject"`

Don't forget to restart Apache after that! You should be able to access the new project over the following URLs:

- `http://localhost/myproject/create_schema.php`
- `http://localhost/myproject/translate_tpl.php`
- `http://localhost/myproject/`

As always with PHP projects, you may add any new files and subdiretories as you like. To ease your life as a developer, most of your work can be delegated to the Athomaris PHP library just by adding the following lines:

- `require_once(“config.php”);`
- `require_once(“$BASEDIR/../common/app.php”);`

and calling only a few functions. Details are explained later. If you want to learn how the default page layout works, inspect `common/generic.php`.

2 Basic configuration

2.1 Basic $SCHEMA configuration

The subdirectory `schema/` contains files of the form `schema`*xx*`.php` where *xx* must be a number. The only purpose of these files is to define different versions of `$SCHEMA` and `$EXTRA`. In this section, only `$SCHEMA` is explained.

`$SCHEMA` is a nested PHP structure.

The outmost level of $SCHEMA is a PHP hash indexed by *table names*, obeying standard identifier syntax. All table names in the system must be globally unique. By default, the PHP table name is also used as SQL table name. This default association can be overridden via the `SQLNAME` attribute (see later).

2.1.1 Table Definitions in $SCHEMA

Each table is described by a nested PHP hash which may contain some of the following attributes:

`SCHEMA_CONTROL` => *boolean*. Defaults to `true` when omitted.
When set to `false`, no SQL statements for schema management will be generated at `create_schema.php`. Use this for external tables where you don't have control over it.

`FIELDS` => *hash* describing the database columns. No default.
This sub-structure is described in section 2.1.2.

`TEMPORAL` => `true` or `false`. Defaults to `true` when omitted.
When a table is created as a temporal table, some special `FIELD`s are automatically added by default. Details are described in section 2.1.3.
When `TEMPORAL` is `false`, no `FIELD`s are automatically added, not even the default primary key *`tablename`*`_id`. Thus you *must* create that explicitly, and you *must* add a `PRIMARY` definition (see below).

`PRIMARY` => *string*. Name of the primary key. Defaults to *`singulartablename`*`_id` when omitted.
Normally this attribute is determined by default. When `TEMPORAL` is `false`, or when interfacing with external databases having different schema conventions, you can set a different name here.
Hint: recommended *best practice* is to use the default only as a *hidden* primary key. This means that the *end*-user should never use its value for any purpose; he/she should not even *see* this value. Instead, the end-user should only see and use `UNIQUE` keys as described below. This has tremendous advantages: defining the user-visible identification attributes as *dependent attributes* allows to change them at runtime. When combined with `REFERENCES` and `"on update cascade"` (see below), your system becomes flexible in a way which is *vital* for large enterprises.
Exception when interfacing to external databases: if you don't have control over the schema (e.g. when connecting to *existing* databases), you can also specify a *combined key* by enumerating the column names separated with comma.

`DB` => *string*. References a database name from `$CONFIG`. When omitted, it defaults to the first entry in `$CONFIG`.
Use this when you maintain multiple database connections in parallel. Details on $CONFIG are in section 2.2.

`ENGINE` => *string*. Only used for MySQL. Defaults to `MyISAM` when omitted.

`UNIQUE` => *array* of strings. Names of secondary keys. No default. Can be omitted.
Tells which columns or column combinations should be uniquely indexed. If you want to create a combined index, just use a string where the column names are comma-separated.

`INDEX` => *array* of strings. Names of indexes. No default. Can be omitted.
Works like `UNIQUE`. The only difference is that non-unique indexes are created.

`ACCESS` => *string* with one of the values ''`r`'', ''`R`'', ''`w`'', ''`W`'', `or` ''`n`''. Defaults to ''`W`'' when omitted.
Maximum access rights for the *whole table* which cannot be exceeded by anybody except `root`. The access right codes have the following meaning:

n no access at all.

r read access to the database, but not displayed to the user GUI.

R read access to both the database and for the user.

w write access to the database, but the user can only read it over the GUI.

W write access for both the database and for the user.

`REALNAME` => *string*. Defaults to the tablename when omitted.
The physical table name as used by the backend database may differ from the *logical* name used by Athomaris. By widely using this feature, you may ease *schema integration* in heterogeneous environments, such as company mergers etc. When combined with the cross-join and cross-referential integrity capabilities of Athomaris, **logical integration** of large systems is possible.

`SINGULAR` => *string*. When omitted, it defaults to the tablename without trailing "s".
The **Athomaris convention** is to use English plural forms for (logical) table names. In contrast, column names should always start with the corresponding singular form whenever they *logically belong*[1] to that table. This way, (generated) SQL code becomes better understandable for humans (and foreign code bypassing the Athomaris PHP library becomes more intuitive). For example, table `foos` uses column names such `foo_id` (here as the primary key). Because there are some linguistic exceptions in English (e.g. "classes" → "class"), you can set the singular form here. The singular is used as a basis for automatic creation of column names, such as `*_id`.
However, when interfacing to external / existing databases, you often have to obey foreign conventions conflicting with the default systematics of Athomaris. In such a case, you can use the following directives for further fine-grained control (the meaning of most of them is described in section 2.1.3):

`FIELDNAME_ID` => *string*. Defaults to ''*`singular`*`_id`''.
This is not necessarily the same as `PRIMARY`, because `PRIMARY` may be a *combined key*, while `FIELDNAME_ID` specifies which column should get an `AUTO_INCREMENT` attribute.

`FIELDNAME_VERSION` => *string*. Defaults to ''*`singular`*`_version`''.

`FIELDNAME_DELETED` => *string*. Defaults to ''*`singular`*`_deleted`''.

`FIELDNAME_MODIFIED_FROM` => *string*. Defaults to ''*`singular`*`_modified_from`''.

`FIELDNAME_MODIFIED_BY` => *string*. Defaults to ''*`singular`*`_modified_by`''.

... NYD

2.1.2 FIELDS

`TYPE` => *string*. No default. Cannot be omitted.
describes the SQL type of the column. Valid types are:

`int, smallint, bigint, int(`*n*`)` Integers (with number of decimal digits n).

`bool, boolean` Integers with allowed values 0 and 1.

[1] Here are some excpetions which don't "logically belong" to some given table:

- Foreign keys (see `REFERENCES` in section 2.1.2): please use the name from the referenced table. This way, almost all possible joins can be expressed as *natural joins*, and the schema is self-documenting for humans.
- Anything which references to external information, such as filesystem data. This way, you explicitly express that the corresponding *domain* is not under your control. If you have many such cases, it is wise to even introduce some systematics for that.

`char(n), varchar(n)` Strings, with lenght or maximum length n.

`datetime, time, timestamp` See corresponding SQL types.

`text, tinytext, mediumtext, longtext` Most databases store variable-length texts columns in a seaparate heap, not in the data record. This affects nonfunctional access properties at runtime.

`blob, tinyblob, mediumblob, longblob` Ditto. In difference to `text`, no comparion for equality is possible.

Note: although some of these types look like being `mysql`-specific, other drivers such as `sybase` will internally convert them to the most closest type available there.

`DEFAULT` => *string*. No default. Can be omitted, but strongy discouraged.
Best practice is to always provide defaults for any column. The default should clearly indicate that initialization has been omitted. This way, you can see when somebody has used the default.
The *string* must denote SQL code. If you want `NULL` as default value, just denote `''null''`. If you want a string, you have to provide the quotes within the string, such as `''''text''''`.

`CHANGE_FROM` => `''`*old_columnname*". No default, should be omitted if not necessary.
Specifies that a column name has changed between two schema revisions (in `schema/schema`*xx*`.php`). In essence, this will be translated to `alter table ... change column ...` statements. Attention! Don't forget to remove the `CHANGE_FROM` assignment in the *following* revision, otherwise it would try to rename the (then non-existing) old column once more!

`REALNAME` => *string*. Defaults to the fieldname when omitted.
The physical column name as used by the backend database may differ from the *logical* name used by Athomaris[2].

`LENGTH` => *array(a, b)*. When omitted, array(0, *maxlen*) is used as default where *maxlen* stems from the `TYPE` definition.
Specifies the minium and maximum allowed string lengths a and b. This is translated to a `check` clause in some SQL dialects, and to a runtime check where SQL is incapable of directly handling this.

`REGEX` => *string*. No default. May be omitted.
Create runtime checks (or in some SQL dialects even `check` clauses) which ensure that the value obeys the given regular expression. The regular expression must be in Perl syntax, enclosed in `''/.../''`.

`REFERENCES` => *hash*. No default. May be omitted.
Speficies that referential integrity between tables must be obeyed at runtime. This works even with MySQL versions which don't support referential integrity such as MyISAM tables (appropriate checking code will be automatically executed by the Athomaris PHP Library at runtime). The *hash* must obey the following structure:

''tablename.fieldname'' => *array(keystrings)*. The *.fieldname* may be omitted when it has the name as the current column (corresponding to a *natural join* possibility). The keystrings must be one or more of the following indicating the well-known ANSI SQL meaning:

```
on delete cascade
on delete set null
on update cascade
on update set null
```

`ACCESS` => *string*. Defaults to `''W''` when omitted.
Specifies maximum access rights for the current column. The meaning of the code letters is the same as already described at 2.1.1. The only difference is the granularity level, which is column granularity here.

[2]In essence, this automatically translates to SQL code like `select` *physicalname* `as` *logicalname*`, ... from ...`

`OPTIONS` => *string*. No default. Should be omitted.
This allows direct throughpassing of MySQL options. Try to avoid this!

`CB_CONV_READ` => *string*. No default, can be omitted.
Hook for supplying a PHP function name which converts the value from a database field into an internal PHP representation upon `db_read()` (see section 5.3.3). The function must have a prototype of the form `convert_read_function($table, $field, $database_value)` and return the internal PHP representation, whatever it may be. Hint: you may use this for arbitrary type conversions, even deeply nested PHP structures. For example, you may provide a function which converts an XML string into a nested DOM structure.

`CB_CONV_WRITE` => *string*. No default, can be omitted.
This hook is the inverse of `CB_CONV_READ`: it just converts in the opposite direction upon `db_insert()`, `db_update()`, etc (see section 5.3.2). The signature of the function follows the same methodology. For example, you can use this for converting a deeply nested PHP structure into an XML string representation.

2.1.3 Temporal tables

When a table `foos` is created with `TEMPORAL => true` (or by default), the following columns are automatically added to the table:

`foo_id` => `array(''TYPE''=> ''bigint'', ''DEFAULT'' => ''auto_increment'')`.
This is the primary key of the table. Best practice is to use this *only* as "physical object id" of data records, but **never** for *logical* identification! Logical identification should always be done via additional secondary keys (see `UNIQUE` in section 2.1.1). This has the major advantage that the logical identification can be *changed* at any time, even consistently across the whole system if you define appropriate `REFERENCES` with `on update cascade` on them (see section2.1.2).

`foo_version` => `array(''TYPE''=> ''timestamp'', ''DEFAULT'' => ''current_timestamp'')`.
The timestamp of any changes to this table is automatically recorded in this column. Thus you may use it for inspection of historical versions of tuples.

`foo_deleted` => `array(''TYPE''=> ''boolean'', ''DEFAULT'' => ''false'')`.
Normally this value is always `false`. When a tuple is *logically* deleted, it is not deleted in reality, but rather *marked* as deleted via this column. This way, you can later inspect even deleted tuples via the temporal table `foos_tp` (see below).

`foo_modified_from` => `array(''TYPE''=> ''varchar(16)'', ''DEFAULT'' => ''null'')`.
This column automatically records the hostname or ip address of anybody who does any modification on some tuple, provided that the access is done via the PHP programming interface such as `db_insert()` (see section 5.3). If you bypass the default programming interface with your own hand-written SQL code, it is highly advisable to provide a value here, because this column is essential for humans if they try to comprehend anything in a complex business application.

`foo_modified_by` => `array(''TYPE''=> ''varchar(16)'', ''DEFAULT'' => ''null'')`.
Automatically records the login username of interactive users, or the scriptname of PHP scripts when they do any modification on some tuple.

When defining a table `foos` as temporal, there will be two tables created behind the scenes:

- `foos` is a SQL view providing the non-temporal view on the table. "Non-temporal" means that you cannot access old versions of a tuple, just as with ordinary flat SQL tables where old values are always overwritten by updates. From a user's perspective, this has (almost) the same semantics as a usual SQL table, as if there were no temporal extensions at all. The only exception is updating: a drawback of MySQL is that it does not allow updates to most views. Therefore you must use `foos_tp` instead of `foos` for updates if you want to bypass the default programming interface with your own hand-written SQL code.

- `foos_tp` is the actual table where all the temporal history is stored. When a tuple is *logically updated* (e.g. via `db_update()` c.f. section 5.3), a *new* tuple is *inserted* into foos_tp behind the

scenes instead (aka COW = Copy On Update strategy). The new tuple has the same `foo_id`, but a newer `foo_version` timestamp.

The relation between `foos_tp` and `foos` is illustrated by the following SQL code:

```
create view foos as
  select * from foos_tp t1 where t1.foo_version in
    (select max(t2.foo_version) from foos_tp t2
     where t1.foo_id = t2.foo_id and not t1.foo_deleted
    );
```

As you can see, this filters out any old versions, as well as any tuples which have been *logically* deleted. However the "deleted" tuples remain accessible via `foos_tp`.

2.1.4 Initializing Data

In order to populate your database with some initial data, you may use the global variable `$INITDATA` (see examples in `demo_basic` and `demo_advanced`). This is handy for startup tasks such as creating initial profiles and users. If you omit initial data for tables `profiles`, `languages`, or `users`, some resonables defaults will be supplied (try `demo_advanced` for details).

`$INITDATA` is a hash, indexed by table names. Each table is associated an array of records. In turn, each record is a hash which associates field names to field values.

Note that initializing is only done on `create_schema.php` (see section 1.1) when there is exactly one `schema/schema`*xx*`.php`. When updating the schema to a newer version, you have to update your data by hand (this is necessary because the initial data may be completely altered in the meantime).

2.1.5 Views

Athomaris can create and manage SQL views in `$SCHEMA`. Views may be built upon arbitrary queries, which may join multiple tables as described in sections 5.3.4 and 5.3.5. A View may be used like any other table, but with some restrictions:

1. Views must not span tables from different $CONFIG hosts (see section 2.2).
2. Views are only readable (equivalent to `ACCESS => ''R''`).
3. There is no primary key (equivalent to `PRIMARY => null`). As a consequence, no `REFERENCES` to views are possible.
4. A view cannot reference to the temporal version of another table, but only to the flat one. Instead of *`tablename`*`_tp`, always *`tablename`* must be referenced.
5. A View may reference other views, provided that referenced views occur *earlier* in `$SCHEMA`. This way, cyclic forward-references are impossible.

A view is constructed by adding the following to `$SCHEMA`:

''*viewname*'' => array(''VIEW'' => ***query***) The schema compiler will automatically determine the corresponding `FIELDS` and other descriptions as you would normally supply in a table definition. You cannot provide other definitions by hand; they will be ignored. The *query* syntax is described in sections 5.3.4 and 5.3.5.

2.2 `$CONFIG` configuration

The variable `$CONFIG` in `config.php` describes the databases of a system and the connection methods for working with them. It is a hash with the following components:

`USE_AUTH` => `true` or `false`. Defaults to `true` when omitted.
When set, the whole system works with profile-based user authentication and authorization. The following tables are automatically added to `$SCHEMA`:

`profiles` A temporal table describing the access rights for a class of users. This table contains the following columns:

`profile_name` Secondary key, uniquely describing each profile.
`profile_descr` Description, for comments and remarks.
`t_`*`tablename`* columns of this form are automatically added and maintained by `create_schema.php` for each other table in the system. It specifies the access rights to the respective table in the same format as described in section 2.1.1. The difference to the specification in $SCHEMA is that here we define *dynamic* access rights for each profile, which may *lower* the rights specified in the *static* `$SCHEMA`. This way, some groups of users may be restricted to readonly access, or be even denied acess to whole tables.
`f_`*`tablename_fieldname`* Automatically added and maintained by `create_schema.php` for all columns of all other tables in the system. It specifies the access rights at column granularity. It may further restrict the access rights as already restricted by `t_`*`tablename`*.

`languages` A temporal table which may be used for internationalization of the user interface. Actually, it just defines which templates to use, so it may be also used for creation of different *skins* (look-and-feel styles) in the same language. It contains the following columns:

`language_name` The secondary key uniquely describing the interface style.
`language_template` The name of the compiled template file to use. Defaults to `generic_generic.php`. Details on the template compiler are in section 3.3.
`language_descr` Description, for comments and remarks.

`users` A temporal table having the following columns:

`user_name` Secondary key, uniquely describing each user.
`user_password` This is stored in encrypted form (see the `password()` function in MySQL).
`profile_name` References `profiles`. Thus it determines the access rights for the user.
`language_name` References `languages`. Thus it determines the user interface for the user.
`user_descr` Description, for comments and remarks.

When `USE_AUTH` is `false`, these tables are neither created nor used. The login authentication process just passes the username and password credentials to the database and hopes to get a connection with the necessary access rights. Since there is no profile info about runtime access permissions, only the `ACCESS` attributes can be used for *static* customization of the UI.

`USE_ENGINE` => `true` or `false`. Defaults to `false` when omitted.
Specifies that additional tables for the business process engine shall be added to `$SCHEMA`. Details are in section 4.

`CONNECTIONS` => *hash*. No default. Cannot be omitted.
Describes the connections to databases. It has the form *`connection_name`* => *hash*, where *`connection_name`* is an arbitrary internal identifier which may be used in `DB` specifications of `$SCHEMA` (see section 2.1.1). The *hash* is two or more associations from the following:

`MASTER` => *string*. No default. Cannot be omitted.
Hostname of the database server.

`DRIVER` => *string*. Defaults to `''mysql''` when omitted.
Currently this is the only driver, but others are planned.

`BASE` => *string*. No default. Cannot be omitted.
Name of the database (see `show databases` command in most SQL dialects).

`USER` => *string*. No default. Can and should be omitted.

`PASSWD` => *string*. No default. Can and should be omitted.
Although you can provide usernames and passwords here, this is considered ***bad practice***. When someone hacks your `wwwrun` account, he can read your passwords in plaintext. Instead, when you omit these parameters here, the login credentials from the Apache authentification is used as database access credentials at runtime. Many system designers try to avoid the overhead of maintaining database access rights for individual users. The Athomaris PHP Library

automates this task: Whenever an entry in `users` is created or updated via `db_insert()` or `db_update()`, the database acess rights (e.g. in MySQL this is the global table `mysql.user`) are automatically updated *in sync* with that. This works even when a `profile` is altered or when the foreign key `profile_name` is changed.

`SLAVES` => *array* of strings. No default. Can be omitted.
When using master/slave replication in MySQL, you can provide a list of slave servers. *Readonly* accesses are directed to (randomly selected) servers from the list, in preference to the master server. This way, the overall system load can be spread more smoothly. This is important in environments dominated by database reads. This feature is crucial for enterprise-grade scalability.

3 Basic Customization

3.1 Link-Headers

By default, each page starts with links to all tables from $SCHEMA. You can customize these and add your own links as follows:

Define a global variable $LINKS (best practice: do this in `config.php`) which is a hash indexed by categories. Each category will later be displayed in a different line. The default links are all in the default category `“Tables:”`. In turn, a link category is another hash indexed by *link names* (see `{textlist}` in section 3.3). In the default links, the link name is equal to the table name; in general you may create you own names as you like. Each link name may be associated with an arbitrary HTML href. For example, the default links for project `demo_basic` are constructed as follows:

- `http://localhost/demo_basic/index.php?table=foos`

Hint: if you want to remove some of the default links, just define something like

- `$LINKS[“Tables:”][“foos”] = “”;`

Of course, by defining a non-empty string you can redirect it so something else. The whole default category `Tables:` can be removed by

- `$LINKS[“Tables:”] = “”;`

Notice that `undef($LINKS[“Tables:”])` will not work because it is equivalent to non-defining `$LINKS` at all, which triggers the default link creation. When you remove the default category, you should provide *all* links by hand, otherwise your application would be completely link-less.

3.2 Generic Application Layer

After inclusion of `common/app.php`, some high-level functions are availabe for easy PHP programming:

`app_get_templates()` This should be called once after you have included the application layer by `require_once(“$BASDIR/../common/app.php”)`. This loads the templates as configured for the currently logged-in user.

`app_links()` When called, the default links as defined by the global structure `$LINKS` (see section 3.1) are displayed.

`app_display_table(“tablename”)` Displays the table in browsable form. When the access permissions allow writing, buttons for manipulation of the data are also displayed.

`app_display_record($table, $cond)` Displays a single record. The selection is specified by `$cond` as described in section 5.3.3. Upon write permissions, an input table is also added.

`app_input_record(“tablename”)` Displays all fields for which the current user has write access, and handles all the corresponding input actions.

`app_display_download($table, $cond, $download_fieldname, $filename)` Starts the download of the contens of an arbitrary field (most useful for BLOBs). This must be called as the only function in a single request, because it generates an HTML header suggesting a filename for saving the data at the client side.

The following request parameters are used by the above functions:

`$_REQUEST["table"]` The table name we are currently working on.

`$_REQUEST["order"]` Only for `app_display_table()`. Specifies a comma-separated list of field names for ordering of the table.

`$_REQUEST["primary"]` Tells which fields are identifying the tuple for `app_display_record()` or `app_display_download()`. This must be comma-separated list of field names. You may specify *any* fields which can *together* uniquely identify the tuple - it need not be the primary key, but can also be an `UNIQUE` key. In addition, you also have to supply field values for these fields. For example, if you specified `$_REQUEST["table"]=''foos''` and `$_REQUEST["primary"]=''foo_id''`, you also have to supply `$_REQUEST[''foo_id'']` in order to uniquely identify the tuple you want to work with.

`$_REQUEST["delete"]` When set, this is a delete request. The tuple is *always* identified by its primary key. Values for the primary key must also be submitted as additional `$_REQUEST` fields. For example, if table `foos` has primary key `foo_id`, the value of `$_REQUEST[''foo_id'']` must be supplied to uniquely identify the record which should be deleted.

`$_REQUEST["change"]` When set, this is a change request for an existing tuple. In addition to values for the obligatory primary key, you can specify as many additional fields as you like to change. When you *omit* a field, its value is left unchanged (see also `db_update()` in section 5.3.2).

`$_REQUEST["insert"]` Like `change`, but a new tuple is always inserted (see also `db_insert()` in section 5.3.2). You should provide the values of all fields; when you omit one, the `DEFAULT` will be inserted instead. Supplying values for the primary key is useless because it is always initialized by `auto_increment`.

`$_REQUEST["edit"]` Tells `app_input_table()` which tuple it should prepare for editing. This does not yet alter the tuple, but just generate a form for altering by the user.

`$_REQUEST["clone"]` Like `edit`, but instead of altering an existing tuple, a new one is presented for editing, inheriting all field values from the original tuple.

<*none*> When neither `edit` nor `clone` is set, a new tuple will be presented for editing with `DEFAULT` input fields.

3.3 Generic Template Mechanism

Templates are used to separate the presentation layer from the business logic. In other frameworks, this is called a "Model-View-Controller" (MVC) architecture. While our model is maintained by `$SCHEMA` / `$EXTRA` and the controller is simply the Athomaris PHP Library invoked by Apache, views (in that terminology) are implemented via our templates.

Templates reside in ASCII files having the extension `.tpl`. The template compiler (see section 1.1) finds all files with that extension and translates them into fast PHP code.

At the outermost level, a template file can contain the following elements:

`{include ''`*`filename`*`'' /}` Includes another file as if its contents were written at the insertion point. After that, you may *redefine* any templates or textlists. Since redefinitions replace the previous definition, you can easily create variants of template sets this way. In particular, new *skins* can be created rather quickly.

`{textlist}...{/textlist}` The content at `...` must be a sequence of ASCII lines, each terminated by a newline. Each line defines a simple macro via the following syntax:

`name` `= substitution_text` The usage of simple macros is explained later (see element `{text ''`*`name`*`''}`). Example: change the default link-header category `Tables:` (see section 3.1) to something else by

- `{textlist}`
- `Tables:  = Hey, here you can browse the following tables:`
- `{/textlist}`

{template ‘‘*name*’’}...{/template} Defines a new template. The template compiler translates this into a PHP function tpl_*name*($data) {...} which can be called like any ordinary PHP function. When called, this function simply outputs the text between {template ‘‘*name*’’} and {/template}. The text may contain further directives and macro expansions (working on the parameter $data) as follows:

{$*var*/} Output the PHP parameter $data[‘‘*var*’’], as provided to the template invocation. If the substitution text contains any HTML special characters, they are quoted to avoid XSS attacks onto your system. Deeper levels of the $data structure may be accessed via the syntax {$*var*->subvar} which outputs $data[‘‘*var*’’][‘‘*subvar*’’]. Note that {$*var*->$*subvar*} would result in something different: $data[‘‘*var*’’][$data[‘‘*subvar*’’]]. As a simple rule of thumb, just remember that the number of $ signs in your code indicates nothing more but the number of substitutions carried out at runtime.

{text ‘‘*name*’’/} Output the simple macro defined in a {textlist}. If *name* does not exist anywhere, its name is outputted unmodified in place of the non-existing definition (fallback). Note there is no quote-protection against HTML code, because the {textlist} should have been written by a trusted programmer who probably even *needs* to include raw HTML code for his purposes.

{text $*var*/} Same as before, but the *name* is (indirectly) fetched from $data[‘‘*var*’’]. The same principle applies orthogonally to all examples where a string in double quotes is used: instead of the string constant, you may always use a variable instead, or vice versa.

{ascii $*var*/} Like {$*var*}, but the text is written in typewriter font (monospaced), blanks are translated to , and newlines are translated into
. This way, a kind of “verbatim output” is produced. Handy for displaying raw computer data which should be protected from HTML interpretation.

{preview $*var*/} Same as {ascii $*var*}, but when the text is very long, it is abbreviated with ... and a link is displayed. When the user clicks on the link, the full text will be displayed in a new window.

{param $*var*/} Like {$*var*}, but URL escaping is used instead of HTML escaping. Especially useful for generating URL parameters like ?paramname={param $myvalue/}.

{raw $*var*/} Dangerous! Like {$*var*}, but bypasses the HTML quote protection. Use only if you really know what you are doing!

{header $*var*/} Generate a raw HTML header. This must be called before any other output (see also PHP function header()).

{row $*var*/} Only for very special cases. Assumes that $data[‘‘*var*’’] is a structured data record. This outputs an escaped string which encodes the whole record, such that it can be decoded again via the library function _tlp_decode_row().

{printf ‘‘*format*’’ $*var*1 $*var2* .../} Obvious semantics for C programmers (see also the PHP function sprintf()). The output is quote-protected.

{var $*name* = *expression* /} Assign a new value to $data[‘‘*name*’’]. The *expression* may be an arbitrary PHP expression which *must not contain any spaces*, and you can access only the variable $data and subfields thereof.

{unset $*name* /} Deletes the variable $data[‘‘*name*’’].

{tpl ‘‘*othername*’’/} Calls the template tpl_*othername* with unchanged argument $data. The template *othername* must exist, otherwise an error is produced. If you want to submit a different argument structure to the callee, you can use one of the following variants:

{tpl ‘‘*name*’’ ($*otherdata*)/} Uses $data[‘‘otherdata’’] instead of $data.

{tpl ‘‘*name*’’ ‘‘*key1*’’ => ‘‘value1’’, ‘‘key2’’ => $*var2*, .../} The callee is provided with new or altered values for the given keys. When you combine this with {if}, you can even do recursive calls.

`{hook ‘‘othername’’/}` Like `{tpl/}`, but no error is thrown if the callee does not exist. This is handy for introducing self-documenting hooks, simplifying plugin architectures.

`{if condition1}...{elseif condition2}...{else}...{/if}` Obvious semantics for PHP programmers. Of course, the `{elseif}` and `{else}` parts may be omitted.

`{loop $var as $value}...{/loop}` The loop body is repeated for each substructure member of `$data[‘‘var’’]`, and `$data[‘‘value’’]` is assigned the corresponing value during the loop. After the loop has finished, `$data[‘‘value’’]` is restored to its previous value.

`{loop $var as $key => $value}...{/loop}` Variant thereof: both `$data[‘‘key’’]` and `$data[‘‘value’’]` are assigned the hash key and the hash value of `$data[‘‘var’’]` during the loop, respectively. Both are restored afterwards.

`{loop as $value}...{/loop}` Variant: by omitting `$var`, you can iterate over the top-level `$data` instead of over one of its substructures. Recommendation: try to avoid this in your designs, because later addition of sibling fields is impossible. Just use an additional dummy level like `$DATA`, even if you currently don’t need it.

3.4 Default Templates

The template mechanism as described in the previous section may be used in almost arbitrary ways. However, application programming can greatly improve efficiency if some appropriate **conventions** are introduced.

The following conventions are used in `../common/app.php` and are recommended as a basis for your own extensions.

3.4.1 Default Template Callbacks

These are very simple: for any template *myname*, another template `before_myname` will be automatically called before its execution, and `after_myname` will be called after it has finished.

Normally, such templates are not defined, thus they will not be called. Notice that you can build even long chains like `before_after_before_myname`, but this is no recommended style.

These callbacks are especially useful for augmenting the default templates with your own extensions.

Note: the easiest way to use Athomaris as an *application framework* (although its primary goal is a *domain framework*) is simply to override some of the default templates with your own version (see `{include/}` in section 3.3).

3.4.2 Common Template Names

`header` Contains the HTML header with `<html>`, `<head>` and `<title>`. Of course, further elements like `<?xml` and `<!DOCTYPE` may also be added, depending on the HTML dialect you plan to use. The default is XHTML. If you want to change the HTML protocol header via `{HEADER/}`, this must come first.

`body_start, body_end` Contains the beginning and closing of `<body>`. This is an interesting hook for injection of your own javascipt code etc.

`footer` Contains the closing of `<html>`.

`styles` This is called from the default `header` template. You may include your own stylesheets or other stuff here. Hint: by using `before_styles` or `after_styles` (see section 3.4.1), you can extend the existing styles without replacing them.

`links` Displays the default links (see section 3.1) as produced by `app_links()` (see section 3.2).

`display_table` Displays a whole table as supplied by parameter `$data[‘DATA’]`. Called by `app_display_table()` (see section 3.2).

`display_subtable` This is internally used for displaying subtables.

`display_record` Called by `app_display_record()`. Shows a single data record.

`display_default` Internally used for generic display of a single entry from a record or a table.

`display_ref, display_reflist` Internally used for display of references to other tables.

`display_*` Further internal templates for display of various items.

`input_record` Called by `app_input_record()` (see section 3.2). Presents input fields for a whole record.

`input_selector` Displays a selector with values from *`fieldname`*`_pool`.

`input_sublist` Displays two boxes, one with values from *`fieldname`*`_pool`, and the other with the selected items. The user may move the items between the boxes and change their order.

`input_subtable` Currently not yet functional.

`input_default` Internally used for input of a single generic input field.

`input_*` Various other input variants, similar in hierarchy to `display_*`.

`tool_search` Displays the default search field for browsing table data.

`tool_page` Displays the default pager for browsing table data.

`tool_history` Displays a checkbox for switching between temporal and non-temporal views of tha data.

`tool_level` Currently not yet functional.

3.4.3 Common Template Variables

By default, the application supplies various information over the `$data` argument of templates. Following are subfields of $data:

`DATA` => *array* of records. This contains the table data for display. The format is the same as from db_read() (see section 5.3.3).

`TABLE` => *string*. The name of the table currently displayed.

`PRIMARY` => *string*. The primary key. A composed key is indicated by a comma-separated list.

`PRIMARIES` => *array* of strings. Each member of composed key as a separate string (useful for `{LOOP}`ing over it).

`UNIQUE, UNIQUES` Like `PRIMARY` and `PRIMARIES`, but indicating the *first* `UNIQUE` key (if one exists).

`ACTION_SELF` => *string*. Contains `$_SERVER["PHP_SELF"]`.

`ACTION_BASE` => *string*. A parameter string referencing the currently running PHP script augmented with all *relevant* GET or POST variables. "Revelant" means that all parameter neccessary for reloading are included, but others are filtered away. You may take this as a basis for creating links, overriding some of the parameters simply by appending other assignments, and so on.

`ACTION` => *string*. Like `ACTION_BASE`, but with additional parameters for all the tools described in section ...

`MODE` => *string*. One of the values `insert`, `delete`, `change`, `new_clone`, or `invalid` indicating the typo of operation currently performed. In case of displaying a tool, the name of the tool like `search`, `page`, or `history` is provided instead.

`PERM` => *array*. Contains a data record, namely the currently active `profile`.

`SCHEMA` => *array*. A copy of the global variable `$SCHEMA`, as produced by the schema compiler.

`EXTRA, EXTRAHEAD` Used for adding extra columns to tables. By default, the buttons for data manipulation are placed here.

4 Business Process Engine

4.1 Concepts

Most business process langugages such as BPEL follow a *procedural* paradigm for describing a workflow, similar to classical procedural programming languages. These models are extremely complex: the current WS-BPEL specification 2.0 prints to 140 pages DINA4 in very small font. In contrast, the Athomaris engine tries to remain simple, but to reach at least the same *expressive power*. In future releases, we want to implement translators from BPEL to our model.

The Athomaris business process engine uses the well-known concept of **finite automatons** as established in computer science for decades. The most important difference to the theoretical concept is **explicit state**: *automatic instantiation* of new automatons, as soon as a new tuple is added to the database. In other words: a new finite automaton is instantiated automatically for *any* tuple of the database (whether it be a newly inserted tuple, or a previously modified tuple), working on that tuple independently from other automatons. The only triggering condition is that some *rule* must match on that tuple. Thanks to implicitly matching *any* existing tuple, the system can achieve almost arbitrary parallelism and scalability. If you want a purely sequential workflow, just don't produce *new* tuples which would be "watched" by the Athomaris engine - instead just update the explicit state which is kept inside your *old* tuple. Another benefit of explicit state is that other applications can query and work with it.

The Athomaris model is much more similar to a rule-based computation paradigm called "blackboard systems". Another similar model is called "coordination models and languages" which has attracted a lot of research some years ago.

Here is a small comparison of concepts between procedural models like BPEL and our non-procedural "descriptive" approach:

model	execution state	parallelism	waiting	control flow	data flow
procedural	implicit	explicit	explicit	explicit	implicit
rule-based	explicit	implicit	implicit	implicit	implicit

Note: this description is for technicians only. We deserve to write another (friendlier) description for business managers, employing their terminology and buzzwords.

4.2 Usage

The business process engine is enabled by `USE_BUSINESS_ENGINE => true` in `$CONFIG` (see section 2.2). This adds some orchestration tables to $SCHEMA as described in the next section. After you have filled those tables which orchestration rules or have updated them, you have to invoke the orchestration compiler via the following URL:

- `http://localhost/demo_business/orchestrator.php`

In a future release of Athomaris, we want to have a graphical tool under this URL where you can view and create orchestrations, probably even with drag and drop. For now, there is only a batchmode compiler.

When compilation is successful, a file `compiled/engine_table.php` is created. Afterwards, you may invoke the business process engine from the commandline by supplying username and password as arguments:

- `cd /www/athomaris/demo_business; php business_engine.php ''root'' ''secret''`

4.3 Tables for the Orchestration Level

Once the business process engine has been enabled via the `USE_BUSINESS_ENGINE` attribute of `$CONFIG` (see section 2.2), the following tables are automatically added to the schema:

`bps` (abbreviation for *business processes*). This table describes the *interface* to a business process. When used standalone without joining to other tables such as `rules`, you get an *abstract* business process (borrowing terminology from other languages such as BPEL). However, when you join this with `rules` and `conts` (see below), you get an *executable* business process. In addition to the usual temporal columns such as `bp_id` and `bp_version`, this table consists of the following columns:

`bp_name` Secondary key. This is used to uniquely identify each business process in the system.

`bp_statefield` Defines on which *cell candidates* the business process should work. Must be a string of the form *`tablename.fieldname`* where *`tablename`* is a valid name of another table from $SCHEMA and *`fieldname`* a valid column name therein.
Currently not yet implemented: if you leave this field empty, a kind of "procedural model" will be executed. State is choosen automatically, but the business process needs to be *called explicitly*.

`bp_inputs, bp_outputs` Not yet implemented. In future releases, this will be used for argument checking when calling procedural-style business processes and sub-processes. Leave it empty for now.

`bp_joinwith` Usually left empty. When you specify a comma-separated list of other table names, these tables will be naturally joined with *`tablename`*. In consequence, you may use their fields in `@{`*`otherfield`*`}` macros (see below). This way you can easily access related data.

`bp_comment` For documentation at the orchestration level.

`rules` This table describes the left-hand part of an automaton rule. In addition to the usual temporal columns such as `rule_id` and `rule_version`, it consists of the following columns:

`bp_name` References `bps`. Tells to which business process the current rule belongs.

`rule_prio` This is used to define an *order* on all rules working for the same business process. In other words, the combined secondary key of `rules` is defined as `UNIQUE => array(''bp_name,rule_prio'')`.

`rule_startvalue` Defines which value a *cell candidate* must have if this rule shall "fire". Once the rule has fired on a candidate, the execution engine remembers the identity of the firing cell (in the following, we mean this single instance when we speak of "cell"). In order to allow multiple modes of testing and matching against cell candidates, the following syntax must be used:

`=`*`value`* Tests the candidate for equality with the given constant.

`%`*`value`* Matches if *`value`* is a substring of the candidate cell.

`/`*`regex`*`/` Matches *`regex`* against the candidate cell. You can use Perl-compatible regular expressions with parentheses. Later these may be used for `@{`*`n`*`}` macro substitutions (see `rule_action` below). Note that full-line matches must be indiated by `/\A...\Z/` or similar.

`rule_condition` Defines additional conditions which must hold if the rule shall fire. The syntax is as follows:

<empty> When left empty, the condition is always true, i.e. there is no additional condition.

`?`*`fieldname=value`* Similar to `rule_startvalue`, but you may test arbitrary other fields. If you have used natural joins at `bp_joinwith`, you can use them here. This is very useful for formulating complex `rule_conditions`. You may denote the same variants of tests as with `rule_fieldvalue`, for example `?`*`fieldname%value`* or `?`*`fieldname/regex/`*.

You may specify multiple conditions on different lines; these are logically anded together. In a future release, we plan to support full-fledged boolean expressions.

`rule_location` Not yet implemented. This will specify the hostname or a group of hostnames where the rule should be executed.

`rule_firevalue` As soon as the rule fires, this value will be immediately written to the cell. This is used for proper restarting in case of system crashes. It simply records the fact that a rule has fired and thus an action has started (and probably must be rolled back). You may provide one of the following syntaxes:

<empty> When left empty, a reasonable default is written into the cell. If the cell contained an integer number, this number is incremented by 1. If it contained a string, the string is *pre*pended with "`start_`". Recommendation: don't use the default, always specify this value explicitly!

`+` ***n*** When starting with a plus sign followed by a blank and a number, this number is added to the old cell contents. Works only well if the cell had an integer type.

<anything else> Will be literally copied to the cell. However, it may contain `@` macros, which are *substituted* as follows:

`@{`***fieldname***`}` This is substituted by the value of ***fieldname***. It may stem from the matching tuple, or from a more complex join delivered by `bp_joinwith`, or from variables set via `var`. When complex nested structures have been delivered by ***fieldname***`_pool` or by a `query` statement, you can access them via the syntax `@{`***field->subfield1->subfield2***`}`.

`@{`***n***`}` When ***n*** is a number, it is substituted by the ***n***-th matching parenthesis of the regular expression denoted in `rule_startvalue`. This way, you can build your own parser and propagate almost arbitrary values from there.

`@(`***subcommand args...***`)` This is substituted with the standard output of ***subcommand***, executed in a subshell. For example, you may do simple calculations such as `@(expr @{foo_state} + 1)`.

`rule_action` Tells what to do when the rule fires. The following kinds of actions are currently implemented (further actions such as SOAP are planned):

`script` ***command args...*** Executes the given command with parameters in a Unix shell. Inside your text, you may use the same macro substitutions as explained earlier. Hint: when starting endlessly running scripts or commands (for example `script socat UDP-RECV:4711 STDOUT` for an UDP listener endlessly collecting messages from many sites on the network), you should either set the `rule_timeout` to 0, or to a very long period, and you should introduce some continuation rules for handling signals or other interruptions of that long-running process.

`url http://...` This is equivalent to `script wget -O - 'http://...'`. You can query arbitrary web or ftp servers this way. When combined with the macro features explained above, you may easily create arbitrary REST queries, just for example.

`var` ***name*** `=` ***'value'*** Set a variable for `@` macro expansion. At the right-hand side, you may use arbitrary macro expansions.

`insert` ***othertablename fieldname1='value1', fieldname2='@{something}'...*** Inserts a new tuple into the database. Notice: when some (other) rules are defined for this table, this effectively results in creating a new non-procedural sub-business-process.

`update` ***fieldname1='value1', fieldname2='@{something}'...*** Update the original candidate tuple the current rule is working on.

`update` ***tablename fieldname1='value1', fieldname2='@{something}'...*** Same, but update a tuple from ***tablename***. When ***tablename*** is mentioned in `bp_joinwith`, you need not supply the primary key because it can be determined automatically. Otherwise, the primary key of the tuple must be explicitly given.

`delete` ***tablename fieldname1='value1'...*** As the name suggests, a tuple is deleted from the database. The primary key must be always given. Be careful with it!

`query` ***varname tablename fieldname1='value1'...*** Query the given table by the given keys and values. The query result is assigned to ***varname*** as an array of data records (see `db_read()` in section 5.3.3).

`call` ***bpsname varname1='value1'...*** Call a sub-business process. The state of the subprocess will be kept in table `states` (see below); its state_value will be set to the constant `'START'`. The variable assignments ***varname1*** etc. may be used for transfering selected values from the caller's environment to the callee's environment. Notice that the caller process will not execute any further `rule_actions` after `call`. Instead, the caller business process will remain in state `rule_firevalue` until the

callee returns.

`start` *`bpsname varname1='value1'...`* Like `call`, but the sub-process is executed *in parallel* to the caller. Unlike call, the caller continues with interpreting actions and continuations und thus may overcome `rule_firevale` until some `cont_endvalue` in parallel to the callee. The callee will ignore any encountered `return` statements and thus behave like a main business process. **Warning:** you can exercise recursive calls and starts, but be sure to include logic for avoiding endless recursion and "fork bombs" among the business processes.

`return` *`'returnvalue'`* Return from a sub-business process to its calling process and set its former statefield (which was stuck at `rule_firevalue` during the call) to *`returnvalue`*. In case the current process was started with `start`, this is a nop.

`return` *`'returnvalue' varname1='value1'...`* Like before. In addition, the selected values are transferred back from the subprocess environment into variables of the caller's environment. This works only if the original caller possessed an environment (as is the case with table `states`).

Multiple actions may be specified, each on a different line. They will be executed sequentially. Note that a failing action (such as a database error) will immediately terminate the action chain - the following actions will not be started upon failure.

Warning: you can use the `script` keyword multiple times at the same rule when given on different lines. However, *each* of the called scripts will produce pseudo events like `START` and `STATUS` (see section 4.4). If you are not careful, this may result in triggering *multiple* actions from following continuations, possibly even after further rules have already fired, resulting in nondeterministic behaviour. This can be quite confusing and probably not what you want! Therefore it is *highly recommended best practice* to use `script` at most once at each rule. If you want to start many actions, just place them into a single `script` statement, separated by semicolons or other shell structures.

`rule_timeout` When non-zero, each started business process is monitored. Whenever it does not respond either on `stdout` or `stderr` within the given timeout (measured in seconds), a pseudo-event `TIMEOUT` is generated (see section 4.4). The pseudo-event can then be handled as described below.

`rule_comment` For documentation at the orchestration level.

Note that table `rules` describes only which business process to start under some preconditions. The *consequences* of a business process execution are solely handled by *continuations*:

`conts` (abbreviation for *continuations*). This table describes the *possible consequences* of a rule execution. In addition to the usual temporal columns such as `cont_id`, it consists of the following columns:

`bp_name, rule_prio` Together they references `rules`. Tells to which rule the current continuation belongs.

`cont_prio` This is used to define an order on all continuations which are *candidates* for handling the same rule invocation. `conts` has the following secondary key: `UNIQUE => array(''bp_name,rule_prio,cont_prio'')`.

`cont_match` Regular expression which matches *each line* from `stdout` and `stderr` of the `rule_action`, as well as pseudo events such as `TIMEOUT` *`n`* or `STATUS` *`n`* (see section 4.4). When the regular expression matches, the continuation is *selected*.

`cont_action` Same syntax as described at `rule_action`. Although it is possible to start another scripts again, their output is not parsed again. **Warning:** use this only for small actions not involving different state changes at the cell. Since the same continuation may be triggered multiple times by different lines of output, you can easily mess up your system with nondeterminism. Be very careful with `cont_action`; try to prefer `rule_action` whenever possible!

`cont_endvalue` After this cont has been selected and the `cont_action` has sucessfully completed, this value is written back into the cell. Used to record the fact that the action has completed. When you leave this empty, no value is written and the next continuation according to `cont_prio` is examined whether this can be selected (continue the candidate selection process). When non-empty, you may use the "+ n" notation and macros as described at `rule_firevalue`.

`cont_comment` For documentation at the orchestration level.

`states` This table holds all internal *runtime* state for sub-business processes when they are created by `call` or `start`. Normally you should not write on this table, because the business process enginge should be the only reader and writer on it. It contains the following fields:

`bp_name` References the name of the caller.

`state_value` Contains the execution state of the sub-process. Upon start, it is inialized with the value `START`. Best practice is to use strings in a systematic way.

`state_env` Contains all the variables of the sub-business process, aka environment.

`state_returnfield` Tells which field of the caller needs to be updated by the *`returnvalue`* of `return`.

`state_returnid` Uniquely identifies the tuple where the *`returnvalue`* needs to be written back.

4.4 Pseudo Events

A pseudo event is generated by the business process execution engine, and it is treated *uniformly* in the same way as a line of output from the script will be treated.

4.4.1 Continuation Pseudo Events

Currently the following events are defined at the continuation level:

`INSERT` *`ok`* Tells whether an `insert` action was successful. 0 means failed, 1 means ok.

`UPDATE` *`ok`* Tells whether an `update` action was successful.

`DELETE` *`ok`* Tells whether a `delete` action was successful.

`DB_ERROR` *`operation`* `(`*`text`*`)` Provides additional cleartext information in case of a database error upon *`operation`* `read`, `insert`, `update`, or `delete`.

`START` *`pid`* `(`*`cmd`*`)` Tells that a `script` executing the commands *`cmd`* has started with process id *`pid`*.

`TIMEOUT` *`pid n`* Tells that the script *`pid`* has not responded in any way for at least *`n`* seconds.

`KILL` *`pid`* The `script` *`pid`* has been sent a `SIGKILL` due to a `rule_timeout` or another error.

`STATUS` *`pid statuscode`* Tells that the `script` has *regularly* terminated with Unix status `exit(`*`statuscode`*`)`. "Regularly" means that its *controlling process* was healthy and able to notice that fact. In the seldom case that even the controlling process died (which never should happen), you should check the global events as described in section 4.4.2.

`SIGNALED` *`pid signalcode`* Reports the Unix signal number in case of a `script` abort.

`FORKED` *`controlpid`* For informational purposes only. Tells that a *controlling process* has been created. The decision whether a controlling process is created is completly up to the engine. Don't rely on it for your business logic. This can be used for finding matching *`pid`*s in global events (section 4.4.2).

`NO_RETURN` Tells that a `return` statemen has been skipped due to a parallel `start`.

4.4.2 Global Pseudo Events

Global events are not associated with any rule or continuation, but rather are associated with the engine itself. Therefore a dummy business process `GLOBAL` must be defined to catch global events. In addition, a rule must be defined with a `rule_prio` of 0. Lastly, you may define as many continuations for `bp_name=''GLOBAL''` and `rule_prio=0` as you like. However, the fields `bp_statefield` as well as `rule_startvalue` and `cont_endvalue` etc are meaningless, because recognition of global events is not executed in the context of a cell. Despite of this, you may issue direct database operations such

as `insert`, `update`, or `delete`, but be sure supply enough information for identifyin the tuples; there is less context information than normally. Don't call `script` again – this may cause endless message pingpong loops – unless you are an expert in Unix process control and *really* know what you are doing.

`ENGINE_ERROR (`*`text`*`)` Provides cleartext information about internal errors from the engine.

`ENGINE_WARN (`*`text`*`)` Provides cleartext information about internal warnings, such as missing macro names etc.

`GLOBAL_STATUS` *`controlpid code`* `(`*`cmd`*`)` The main process checks regularly whether any of its controlling processes *`controlpid`* have terminated with status *`code`* while spawning further subprocesses for *`cmd`*. Notice that this is different from the continuation `STATUS` message: STATUS tells what's the matter with a `script` subprocess, while `GLOBAL_STATUS` tells what's the matter with the *controlling process* for the script.

`GLOBAL_SIGNALED` *`controlpid signalcode`* `(`*`cmd`*`)` Similar to `GLOBAL_STATUS`, but reports the Unix signal number which caused the abort.

5 Advanced Features

5.1 Automatic Data Synchronization

...

5.2 Subrecords and Display/Editing of References

...

5.3 Programmers API to the Database

This section is devoted to experienced PHP programmers for whom the default `app.php` is not enough, or if they like to program their own interfaces or complex database access engines.

The purpose of the following functions is abstracted access to temporal tables, which is independent from SQL dialects.

5.3.1 Data Format

All database operations use a common PHP representation for table data, called `$data`. It is simply an *array of records*. A record is simply a hash, indexed by the column names. For example:

```
$data =
  array(
        array(
              ‘‘foo_id’’ => 17,
              ‘‘foo_comment’’ => ‘‘first_example_record’’,
             ),
        array(
              ‘‘foo_id’’ => 18,
              ‘‘foo_comment’’ => ‘‘next_example_record’’,
             ),
       );
```

As you can see, the column names are *deliberatly* repeated. Although OO classes *could* save *some* of that runtime space overhead *in theory*, maintaining OO class definitions instead would be a greater effort, and it would lower flexibility. In particular, `ACCESS => ‘‘n’’` (see section 2.1.1) will omit inacessible fields *dynamically at runtime*, individually for each field. Another use is for `db_update()`, see section 5.3.2: There you can simply specify *which* fields to update, just by omitting all those fields you don't want to change. Note that in absence of database transactions, this has different *atomic properties* from first reading a full tuple, updating the PHP structure, and finally writing back the full tuple. The difference becomes clear with MySQL MyISAM tables in presence of *concurrent* updating.

This kind of flexibility is hardly to achieve with the usually static OO class concepts. Thus we use a “functional” programming style (inspired by languages like ML, Haskell or good old Lisp), and we pay some (acceptable) runtime space overhead in favor of a simple design. Nevertheless, future releases of the Athomaris PHP library could support automatic generation of class definitions from `$SCHEMA`.

5.3.2 Update Operations

`$ok = db_insert(‘‘table’’, $data);` This will insert all the records from `$data` (see section 5.3.1) into the named table. Atomicity is only guaranteed for the consisteny of each tuple; however

depending on the database type and the network distribution, atomicity for the whole set of records could be nethertheless achievedor at least approximated.

`$ok = db_update(“table”, $data);` This will update all *present* fields of all records from `$data`, with one exception: all fields from the `PRIMARY` key *must* be present and are *not* updated; otherwise the tuples could not be uniquely identified.

`$ok = db_replace(“table”, $data);` When the tuples are not yet present in the databases, this will lead to the same effect as `db_insert()`. Otherwise, the effect of `db_update()` will be achieved.

5.3.3 Reading of Data

`$data = db_read(“`*table*`”, “`*field1*`,`*field2*`,...”, $cond, $order, $start, $count);`
This will retrieve data from the named table. The parameters have the following meaning:

“*field1*`,`*field2*`,...”` When this parameter is empty, all *accessible* fields (at least `ACCESS => “r”`, see section 2.1.1) will be retrieved.

`$cond` When empty or `null`, all tuples from the whole database are retrieved. In its simplest form, you may provide a hash `array(“`*fieldname*`” => `*fieldvalue*`)` with the obvious meaning: only tuples exactly matching the *fieldvalue* are selected. When providing multiple hash keys, all conditions are logically anded. More complex boolean expressions may be constructed in the following way:

1. the variant “*fieldname operator*” `=>` `fieldvalue` uses *operator* instead of the default equality. Following *operator*s are possible:

`= <> < > <= >=` The usual comparison operators.

`%` Tests for `like` as defined by ANSI SQL.

`rlike` Tests for MySQL `rlike`. Not portable to other databases! Try to avoid this.

`!` Tests whether *fieldname* is SQL `NULL`. The *fieldvalue* is ignored.

`@` Tests whether *fieldname* is *not* SQL `NULL`. The *fieldvalue* is ignored.

`in` The *fieldvalue* must be a sub-query as described in sections 5.3.4 or 5.3.5. The meaning is intuitively clear to SQL programmers.

`exists` The *fieldname* must be empty, and the *fieldvalue* must be a sub-query as described in sections 5.3.4 or 5.3.5. The meaning is intuitively clear to SQL programmers.

`not exists` Similar; the meaning is intuitively clear to SQL programmers.

1. when *fieldvalue* is an *aggregating sub-query* as described in section 5.3.5, a nested natural join with the *table* is computed (resulting in a *dependent* subquery), and its result is compared with *fieldname*. An operator may be appended to *fieldname*.

2. when “*fieldname*”`=>` is omitted and *fieldvalue* is itself a hash obeying the same construction rules, these conditions are logically *ored* instead of anded. When recursively nesting such expressions, oring and anding will always *alternate,* depending on odd or equal nesting level.

`$order` When empty, the result tuple may be unordered. Otherwise it must be a *string*, containing a comma-separated list of fieldnames. Alternatively, it may be an array of plain fieldnames.

`$start` When empty or 0, the result set is starting at its beginning. Otherwise it must be a number indicating the start position.

`$count` When empty or 0, the number of tuples is not restricted. Otherwise at most `$count` tuples will be delivered.

5.3.4 Full Queries / Subqueries

Queries and subqueries may be uniformly expressed as a single *hash* containig the following keys:

`TABLE => ''table''`. Specifies the table to query. When a comma-separated list of table names is provided, the *natural join* of all tables is computed.

`FIELD => $struct`. When empty, all fields are retrieved. When `$struct` is a plain string containing a comma-separated list of fieldnames, only those fields are retrieved. When it is a hash, the following variants are possible:

''*fieldname*'' Retrieve that field.

''*aliasname*'' `=> $subquery` The subquery must be recursively structured in the same way as described in this section. The inner query is automatically *dependent* from the outer one by all common fieldnames (similar to a *natural join*), and the result of the subquery is propagated to the outer query under the name *aliasname*. This way, you may create "virtual fields" containing *dynamically computed* values.

`COND => $cond`. Query condition, as already described in section 5.3.3. Recursive nesting is possible.

`ORDER => $order`. Already described in section 5.3.3.

`START => $start`. Already described in section 5.3.3.

`COUNT => $count`. Already described in section 5.3.3.

`JOINFIELDS => ''fieldname1,fieldname2,...''`. May be omitted. Defaults to *all* fieldnames occurring in more than one table from the `TABLE` list.
Use these fields for constructing a natural join. Each of the fieldnames must occur in at least two of the tables mentioned by `TABLE`.

`JOIN_ON =>` *array* of strings. May be omitted. Defaults to *all* possible conditions derivable from `JOINFIELDS`, expressed in the following syntax:

''*tablename1.fieldname1=tablename2.fieldname2*'' The tablenames must occur in `TABLE`, and each fieldname must occur in the corresponding table. When *tablename1* and *tablename2* are different, non-natural joins may be expressed. When you provide `JOIN_ON`, `JOINFIELDS` is ignored as a whole. By supplying an empty array, you may express the full cartesian product between the tables.

Instead of calling `db_read()`, you may use `_db_read()` the following way:

```
  $data = db_read(''table'', ''field1,field2,...'', $cond, $order, $start,
$count); is equivalent to $data = _db_read(array(''TABLE'' => ''table'', ''FIELD''
=> ''field1,field2,...'', ''COND'' => $cond, ''ORDER'' => $order, ''START'' => $start,
''COUNT'' => $count);
```

5.3.5 Aggregated Queries / Subqueries

The keys `TABLE`, `COND`, `ORDER`, `START` and `COUNT` have been already described in the previous section. Instead of `FIELD`, you provide the follwing key:

`AGG => $struct`. This is another *hash* consisting of the following two members:

`FIELD => ''expression''`. Allowable is `min(*)`, `max(*)`, `count(*)` and `avg(*)`.

`GROUP => ''fieldlist''`. This specifies the fields over which aggregation will be performed. The meaning is intuitively clear to SQL programmers knowing `group by` statements.

In addition, `JOINFIELDS` or `JOIN_ON` may be added as described in section 5.3.4.

5.3.6 DB Callbacks for PHP Programming

The following hooks may be defined either in `$SCHEMA`, or they may be defined in `$EXTRA`. `$EXTRA` is meant for application-specific extensions which might vary among different applications accessing the same core schema. Following are table attributes:

`CB_BEFORE_INSERT` => *string*. Name of the PHP function to call before `db_insert()` really inserts new tuples, but after basic checks such as for referential integritiy have been successful. The called PHP function must have the following prototype:

`called_function($table, $data)` In essence, the data is the same as supplied to db_insert(), but some changes such as automatic filling of fields like *`table`*`_modified_from` (see section 2.1.3) are already applied. The function **must** return the data which shall be inserted. If you return an emtpy array, nothing will be inserted. You may use this for implementing arbitrary filters, or for doing advanced kinds of referential integrity by hand.

`CB_AFTER_INSERT` => *string*. Same as before, but the hook will only be called after successful insertions.

`CB_BEFORE_UPDATE, CB_BEFORE_REPLACE, CB_BEFORE_DELETE` => *string*. As the name suggests, these are called before `db_update()`, `db_replace()`, or `db_delete()` executes the respective operation.

`CB_AFTER_*` In the same systematics, these are called *after* the respective operation.

`CB_BEFORE, CB_AFTER` These hooks are called whenever *any* of insert, update, replace, or delete are executed.

5.4 Application callbacks

The following hooks are called by the application layer in `common/app.php`. They may be defined either in `$SCHEMA` or in `$EXTRA`.

Following are callbacks at the table level:

`CB_SUBMIT` This is called whenever the user submits tuple data via HTPP GET or POST operations. The format is as always with data callbacks:

`called_function($table, $data)` The `$table` is known from the HTTP parameter `table=`*`tablename`*, while the table data is supplied by the user. The function **must** return the data for further processing.

Following are callbacks at the `FIELDS` level:

`CB_DOWNLOAD` This hook is called whenever the user clicks on a download link.

A FAQ

Q: Is Athomaris really a "framework"?

A: Many people think that a "framework" must be object-oriented (OO). However, most definitions of "framework" don't require that[1]. Athomaris can be classified as a (non-object-oriented) **domain framework** (in contrast to the more widely-known *application frameworks*) for the very general domain of SQL databases, business processes, and web interfaces.

Q: What's the difference to Hibernate (or other OR mappers)?

A: OR mappers transform from some particular paradigm to another one: from the *relational data model* of SQL to the *OO programming language* model. Although some people don't like to hear it, I'll say it: this transformation is *lossy*. The relational *data model* is **more powerful** than the OO one (see also some papers from Codd from the 1970s which I *really* recommend to read). For example, high-level symmetric *join opportunities* are replaced with asymmetric low-level *pointers*. In contrast to object-relational (OR) mappers, Athomaris stays at the SQL model and even extends its power with additional features such as temporal tables.

Q: Why isn't Athomaris object-oriented?

A: Well. Why should it be? `;)`

Seriously, Athomaris' design bears some principles from the **functional programming paradigm** (e.g. found in languages like ML, Haskell, Lisp etc). Where necessary, callback functions are passed as arguments to other functions.

Ok, you can do that also with OO. So why not OO? Because conventional OO *type systems* require some **overhead** for maintaining class definitions: any time something in `$SCHEMA` changes, new class definitions for the *programming language* ("syntactic sugar") must be generated. Even more worse, *code* for accessing the data elements must be generated and maintained (Java, for examples, spends a lot of effort on *accessors*). If Athomaris would copy that method, dynamic runtime evaluation of our `ACCESS` attributes and profile support would be much more difficult. Can you tell the "type" of a table when in situation A some columns are readonly or even inaccessible, but in situation B another *disjoint* set of columns has to be used? Notice that a "missing" field should not even *tried* to be read (which is *different* from reading `null` values, since `NULL` is a valid SQL value). We don't want to generate class definitions for each member of the full cartesian product of all `FIELD`s with all possible `ACCESS` permissions. Thus we take an approach different from contemporary OO styles: we take the "type" as a dynamic runtime *parameter*, and we implement some kind of *"interpreter"* dealing with "types" at runtime. Current OO type systems would *hinder* that because of their static nature.

Q: Are you really sure that Athomaris is *not* object-oriented?

A: No. I'm not sure. Athomaris uses some *concepts* from OO, but it does not employ conventional OO *programming style*. Ok?

Ok, what are these OO concepts? See my paper "On variants of genericity" (availabe via Google). While the interpreter for the "type system" uses *universal genericity*, the template system employs *extensional genericity* simply via `{include/}` and *overwriting* of old definitions. As explained in my paper, the latter is also used by OO inheirtance at the *conceptual* level.

Q: Will Athomaris have future OO extensions?

[1] A common wide-spread definition: a software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.

A: Yes. Although it *internally* uses a functional paradigm, future releases will generate class definitions. Then you can choose whether you get an array of data records out of `db_read()`, or an array of OO objects.