Technical Manual of the `Ceylan-Myriad` Layer

Organisation: Copyright (C) 2008-2020 Olivier Boudeville

Contact: about (dash) myriad (at) esperide (dot) com

Creation date: Sunday, August 17, 2008

Lastly updated: Thursday, August 13, 2020

Status: Stable

Version: 1.2.13

Dedication: Users and maintainers of the `Myriad` layer.

Abstract: The role of the Myriad layer (part of the Ceylan project) is to gather all Erlang general-purpose base constructs that we found useful for (Erlang-based) developments.

We present here a short overview of these services, to introduce them to newcomers. The next level of information is to read the corresponding source files, which are intensely commented and generally straightforward.

Table of Contents

Overview & Context

When using any programming language, there are always **recurring patterns** that prove useful.

Instead of writing them again and again, we preferred gathering them all in a **low-level layer** (mostly a modest **code library**), in their most convenient, reliable, efficient version, together with their specification, documentation and testing.

This layer provides its (generally lightweight, simple) services just on top of the Erlang[1] language, as a relatively small (comprising currently about 70k lines), thin layer.

These services tend to stay away from introducing any new dependency. Should a key, generic service need a third-party prerequisite (ex: library to manage a complex data format, or to process specific data), that dependency should be made fully optional[2] (then, should that dependency be found not available at build or run time, the corresponding service would be transparently disabled).

As a consequence, for the Ceylan project, the first level of the Erlang-based software stack that we use relies on this `Myriad` layer - whose official, more specific name is the `Ceylan-Myriad`[3] layer.

Usage Guidelines

License

The `Myriad` layer is licensed by its author (Olivier Boudeville) under a disjunctive tri-license, giving you the choice of one of the three following sets of free software/open source licensing terms:

- the Mozilla Public License (MPL), version 1.1 or later (very close to the former Erlang Public License, except aspects regarding Ericsson and/or the Swedish law)
- the GNU General Public License (GPL), version 3.0 or later
- the GNU Lesser General Public License (LGPL), version 3.0 or later

This allows the use of the `Myriad` code in as wide a variety of software projects as possible, while still maintaining copyleft on this code.

Being triple-licensed means that someone (the licensee) who modifies and/or distributes it can choose which of the available sets of licence terms he/she is operating under.

Enhancements are expected to be back-contributed (hopefully), so that everyone can benefit from them.

[1] If needing to discover/learn Erlang, we recommend browsing Learn You Some Erlang for great good! or, even better, buying their book!

[2] One may refer for example to what we did respectively for HDF5 and for JSON parsers in the context of REST support, with the `USE_HDF5` and `USE_REST` Make variables.

[3] It was formerly known as the `Common` layer.

About Layers

The `Myriad` services are to be used by this layer itself (for its inner workings), and, more importantly, are to be re-used, specialised and enriched by layers built on top of it.

The general rule is that a layer may depend on (i.e. make use of) all layers *below* it (not only the one just preceding it), but cannot refer to any layer *above* it (it should be literally unaware of their existence).

So, in a bottom-up view, generally a software stack mentioned here begins with the operating system (typically GNU/Linux), then Erlang/OTP, then `Myriad`, then any layer(s) built on top of them (ex: WOOPER).

Of course a given layer does not mask the layers below; for example programs using the `Myriad` layer typically use also a lot the services brought by the Erlang base libraries.

Recommended Usage & Contribution

When developing Ceylan-based code, if needing a service already provided by this `Myriad` layer, it is strongly advised to use that service and, possibly (if useful), expand or enrich it, with backward compatibility in mind.

If such a service is not provided by the current version of the layer, yet is deemed generic enough, then it should preferably be added directly to the relevant part of the library and called from the code that was needing it.

Of course, contributions of all sorts are welcome.

We do our best to test, at least lightly, each element provided. All services offered in a `foo.erl` file are thus expected to be tested in the companion `foo_test.erl` file, in the `test` tree (whose structure tends to mirror the one of the `src` tree). Once there, running this test is as simple as executing:

```
$ make foo_run
```

Note that however we have not reached the discipline level of an exhaustive `eunit` test suite for each service (most of them being almost trivial).

The Dialyzer static analysis tool is regularly run on the code base (see the `generate-local-plt` and `self-check-against-plt` generic Make targets for that).

Getting Myriad

Prerequisites

The **operating system** is supposed to be any not-so-old `GNU/Linux` distribution[4].

People reported uses of Myriad on `macOS`, yet no extensive testing has been done there.

Whereas Erlang supports `Windows` and we tried to be as cross-platform as possible, even with tools like `MSYS2` / `MinGW-w64` we suppose quite a lot of special cases would have to be addressed (patches welcome, though!).

The main tool prerequisite is of course having the Erlang environment available, in its `23.0` version[5] or more recent.

[4]For what it is worth, we prefer Arch Linux, but this does not really matter here.

[5]Most probably that older versions of Erlang would be more than sufficient in order to build Myriad (possibly at the expense of minor changes in a few calls to standard modules having been deprecated since

There are various ways of obtaining it (from your distribution, from prebuilt packages, directly from the sources), one of which being the install-erlang.sh script that we devised.

A simple use of it is:

```
$ ./install-erlang.sh --doc-install --generate-plt
```

One may execute `./install-erlang.sh --help` for more guidance about how to configure it, notably in order to enable all modules of interest (`crypto`, `wx`, etc.).

Getting Myriad's Sources

This is pretty straightforward, based on the project repository hosted by Github:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad.git myriad
```

This should download in your current directory the full Myriad repository. For OTP compliance, using for such a clone its short name (`myriad`) rather than its long one (`Ceylan-Myriad`) is recommended.

The Myriad `master` branch is meant to stick to the latest stable version: we try to ensure that this main line always stays functional (sorry for the pun). Evolutions are to take place in feature branches and to be merged only when ready.

Building Myriad

If a relevant Erlang installation is available, this is as simple as:

```
$ cd myriad
$ make all
```

The parallel build of the whole layer (services and tests alike) shall complete successfully (if it is not the case, see our support section).

One may just run `make` by itself in order to list the main available options.

One may run `make create-myriad-checkout` in order to create, based on our conventions, a suitable `_checkouts` directory so that rebar3 can directly take into account local, directly available (in-development) dependencies (although Myriad does not have any, beside Erlang itself).

Testing Myriad

Just run, still from the `myriad` directory:

```
$ make test
```

The testing shall complete successfully (if it is not the case, see our support section).

then). It is just that in general we prefer to stick to the latest stable versions of software such as Erlang, and advise you to do so.

Type-checking Myriad

As Myriad is (by default) to enable debug information with a key-based protection of the resulting BEAM files, one should first have such key defined.

One way of doing so is, if wanted, to update the default key (see `DEBUG_INFO_KEY` in `GNUmakevars.inc`) and to write in on disk (ex: `make write-debug-key-file`), and to rebuild Myriad accordingly afterwards (ex: `make rebuild`).

Then, still from the `myriad` root directory:

```
$ make generate-local-plt self-check-against-plt
```

It will trigger a full type-checking of Myriad, done thanks to Dialyzer.

This time-consuming phase will complete with a rather long list of notifications. Help us reducing it! These messages are numerous, but we do not think that most of them are so worrying.

Finally, to trigger in one go a full rebuild, testing and type-checking, one may run:

```
$ make check
```

OTP Build

We felt that OTP build tools and Emakefiles were not expressive enough for our needs: as mentioned in Building Myriad, a full, rather complete/complex/powerful build system based on GNU make is used by Ceylan-Myriad natively instead.

It allows to introduce all the generic rules we wanted, to define many conditional settings, to walk through an arbitrarily nested source tree, to integrate within a layered stack (notably alongside some other `Ceylan-*` libraries that depend on Ceylan-Myriad) and to perform a multi-stage build to accommodate the compilation and use of parse-transforms, with their own set of prerequisites.

However, to better integrate with other Erlang developments (which are mostly OTP-compliant), we added the (optional) possibility of generating a Myriad *OTP library application* out of the build tree, ready to be integrated into an (OTP) *release* and to be available as an Hex *package*. For that we rely on rebar3, relx and hex.

OTP Application Myriad is not an *active* OTP application, and as such does not rely on, or provides, services running in the background; so no supervision tree or `gen_server` is involved here, just a *library* application ready for OTP integration[6].

Getting rebar3 There are various ways for obtaining `rebar3`; we prefer:

```
$ cd ~/Software && git clone https://github.com/erlang/rebar3.git
    && cd rebar3 && ./bootstrap
```

Alternatively, should you just want to update a (pre-existing) rebar3 install, first get the current version (`rebar3 -v`) to check it afterwards, then issue `rebar3 local upgrade`; however this would involve running rebar from `.cache/rebar3/bin`, so instead we prefer using (typically from `~/Software/rebar3`):

```
$ git pull && ./bootstrap
```

[6]Speaking of OTP, in development mode, `proc_lib`-based spawns used to be enabled, yet this led to longer error messages that were not that useful; see `spawn_utils.hrl` if wanting to re-enable them.

Another option is to download a prebuilt version of rebar3.

Finally, one may prefer using the install-rebar3.sh script that we devised, which automates and enforces our conventions while letting the choice between an installation from sources or from a prebuilt version thereof (just un `install-rebar3.sh --help` for guidance).

Generating Ceylan-Myriad Then, from the root of a Myriad clone, to obtain the Ceylan-Myriad library *application*, one just has to enter:

```
$ make rebar3-application
```

It will trigger `rebar3`, resulting[7] in a full, OTP-compliant build tree created in `_build` (including a properly-generated `_build/default/lib/myriad/ebin/myriad.app` file), and more generally in a proper OTP application.

Testing Ceylan-Myriad As a result, the OTP application support can be tested from the root of an (already-built, with `make rebar3-application`) Myriad source tree:

```
$ cd src/utils
$ make myriad_otp_application_run
       Running unitary test myriad_otp_application_run (third form) from
          myriad_otp_application_test

--> Testing module myriad_otp_application_test.

Starting the Myriad application.
Myriad version: {1,0,11}.
Current user name: 'stallone'.
Stopping the Myriad application.
Successful end of test of the Myriad application.
=INFO REPORT==== 18-Jul-2019::22:37:24.779037 ===
   application: myriad
   exited: stopped
   type: temporary

--> Successful end of test.

(test finished, interpreter halted)
```

This support can be also tested manually, directly through the build tree used by rebar3; from the root of Myriad, after having run `make rebar3-application`:

```
$ erl -pz _build/default/lib/myriad/ebin/
Erlang/OTP 22 [erts-10.4] [source] [64-bit] [smp:8:8] [...]
```

[7]The operation was previously done through a rebar pre-compile hook, so that the our native build system could be relied upon before injecting the produced BEAMs into rebar's `_build` tree. Because of extraneous, failing recompilations being triggered by rebar, now we rely on a build system parallel to - and directly inspired by - our native one, directly done from within rebar (once properly triggered by our user-oriented Make targets).

```
Eshell V10.4  (abort with ^G)
1> application:start(myriad).
ok
2> text_utils:format( "Hello ~s", [ world ] ).
"Hello world"
3> application:stop(myriad).
=INFO REPORT==== 18-Jul-2019::22:47:36.429804 ===
   application: myriad
   exited: stopped
   type: temporary
```

When needing to include a Myriad header file (taking `spawn_utils.hrl` as an example) in one's code, OTP conventions mandate using:

```
-include_lib("myriad/include/spawn_utils.hrl").
```

rather than:

```
-include("spawn_utils.hrl").
```

OTP Release Quite similarly, to obtain a Ceylan-Myriad OTP *release* (relx being used in the background), possibly for a given profile like `default` (development mode) or `prod` (production mode) - refer to `REBAR_PROFILE` in `GNUmakevars.inc`, one just has to run, from the root of Myriad:

```
$ make rebar3-release
```

Hex Package The hex package manager relies on mix, which is commonly installed with Elixir (another language built on top of the Erlang VM).

Thanks to the rebar3 integration with the `rebar3_hex` plugin specified in Myriad's (generated) rebar.config, `hex` will be automatically installed and set up.

By following the publishing guidelines ([1], [2]), we were able to publish Hex packages for Myriad that can be freely used. And there was much rejoicing!

One just has to specify for example `{deps,[myriad]}.` in one's `rebar.config`, and that's it.

For more details, one may have a look at:

- rebar.config.template, the general rebar configuration file used when generating the Myriad OTP application and release
- rebar-for-hex.config.template, to generate a corresponding Hex package for Myriad (whose structure and conventions is quite different from the previous OTP elements)
- rebar-for-testing.config.template, the simplest test of the previous Hex package: an empty rebar project having for sole dependency that Hex package

Other OTP-related Make Targets of Interest To populate/update the OTP build tree (by default, from the GIT root, for example `_build/default/lib/myriad/` for Myriad) of the current Ceylan layer, one may use:

```
$ make rebar3-compile
```

(this is especially useful in order to be able to use directly, from an OTP application, changes just performed in a Ceylan-based layer)

To update both the OTP build tree and the local ebin directory of each Ceylan layer on which the current layer depends, use:

```
$ make rebar3-local-update
```

(note this will be a no-op from Myriad, as it does not depend on any Ceylan layer)

To publish an Hex package (once the proper version number has been set in `GNUmakevars.inc`, see `MYRIAD_VERSION`):

```
$ make rebar3-hex-publish
```

To test such a package:

```
$ make test-hex-package
```

To populate directly the OTP local build tree with the Ceylan dependencies located alongside the current install (not useful for Myriad - which depends on none, but useful for upper layers) rather than fetching them through Hex (otherwise may more Hex packages would have to be published for testing during development):

```
$ make rebar3-local-update
```

Many more targets are defined in GNUmakerules-explicit.inc.

Myriad Builder

The Myriad is one way of building simply, automatically and reliably an Erlang-based application.

Purpose Initially all our Erlang developments were built using an ad-hoc, Make-based system. It worked great, and is still working perfectly satisfactorily.

A lot of people promote the OTP framework and conventions, and after we had a server-based application to deploy (namely US-Web)

rebar3, relx, hex

with due respect to all the kind people writing documentation and tools I felt we ended up with an overly complicated fairly fragile offering few services at the expense

not much support https://www.rebar3.org/discuss/5ec58fa5ffbfc3018ed2eab9

nor guarantee of durable maintenance

I would spend less time writing my own tool than struggling with the issues of these ones.

WOOPER advanced build

Services offered by the `Myriad` Layer

The `Myriad` services are gathered into following themes:

1. General build structure
2. General settings
3. Maths services
4. Data-management services
5. Helpers for graphical user interface (GUI) programming
6. All-purpose helper scripts
7. Utility toolbox
8. Management of units
9. Metaprogramming, based on heavy use of parse transforms
10. SQL support

In future versions of this document, following topics will be discussed:

- HDF5 support
- REST support
- third-party language bindings (ex: Python, Java, maybe in the future Haskell; C/C++ is to be tackled by our Seaplus project)
- RDF support

Even if this document does not constitute an exhaustive walk-through, each of them is detailed in turn below.

The next level of detail is to peer at the referenced source files, as they include many implementation notes, comments and typing information.

General Build Structure

Various elements are defined at the `myriad` level to set-up an appropriate build, based on GNU Make.

This includes:

- a set of pre-defined Make **variables**, describing various settings that will be reused by generic rules (ex: to compile modules with relevant flags, to create source archives, to install an application, to manage the various paths, to perform test checking, to generate archives, installs and releases, etc.); these variables are defined in `myriad/GNUmakevars.inc`
- a set of generic **rules**, to compile and run various modules and tests, to generate various elements of documentation, etc.; these rules are defined (still from the `myriad` root directory), in:
  - `GNUmakerules-automatic.inc`, for all rules that apply generically to a some kinds of targets
  - `GNUmakerules-explicit.inc`, for all "direct" rules, that are not pattern-based
  - `doc/GNUmakerules-docutils.inc`, for all documentation-related rules
- finally, the whole is gathered in a unique file to include, `GNUmakesettings.inc`, whose structure allows for a safe and sound combination of all these build element across a series of layers (the first of which being Myriad)
- **examples** of minimal Make files, which mostly specify the relative base path and only refer to the generic variables and rules; see `myriad/src/GNUmakefile` as an example

These build facilities are designed to be enriched in turn by all layers above, which may add or override variables and rules.

An example of this stacked structure is the `Ceylan-WOOPER` layer (see official site), which is directly built on top of `Ceylan-Myriad` (and itself a base layer for other layers and applications).

General Settings

These general-purpose settings and helpers, gathered in the `myriad/conf` directory, deal with:

- default CSS files (`Default-docutils.css`)
- our recommended versions of (commented) configuration files for various tools:
  - for `Emacs`: `init.el`, to be placed in the `~/.emacs.d/` directory
  - for `Nedit`: `nedit.rc`, to be placed in the `~/.nedit/` directory
- our standard script to properly install Erlang (`install-erlang.sh`) with detailed comments and command-line options (use `install-erlang.sh --help` for more information)

Maths Services

Some simple maths-related operations are defined in the `myriad/src/maths` directory:

- the most basic services are centralised in `math_utils.erl` and provide:
  - **general operations** apparently lacking to Erlang (for example for conversions or rounding (`floor/1`, `ceiling/1`), or not exactly implemented as we would have liked (ex: `modulo/2`)
  - operations tailored to operate on **floating-point values** (ex: `are_close/{2,3}`, `are_relatively_close/{2,3}`, `get_relative_difference/2`, `is_null/1`)
  - operations on **angles** (ex: `radian_to_degree/1`, `canonify/1`)
  - the associated **typing** information
- linear-related operations are defined; for example the **2D** operations are defined in `linear_2D.erl` (their **3D** counterparts being defined in `linear_3D.erl`, their **4D** counterparts in `linear_4D.erl`; base ones in `linear.erl`) and include:
  - operations on **points**: `are_close/2`, `is_within/3`, `square_distance/2`, `distance/2`, `cross_product/2`, `roundify/1`, `get_integer_center/2`, `get_center/2`, `translate/2`, etc.
  - operations on **vectors**: `vectorize/2`, `square_magnitude/1`, `magnitude/1`, `scale/2`, `make_unit/1`, `normal_left/1`, `normal_right/1`, `dot_product/2`, etc.
  - operations on **lines**: `get_line/2`, `intersect/2`, `get_abscissa_for_ordinate/2`, etc.
  - operations related to **angles**: `is_strictly_on_the_right/3`, `is_obtuse/1`, `abs_angle_rad/3`, `angle_rad/3`, `abs_angle_deg/3`, `angle_deg/3`, etc.
  - operations on **sets of points**: `compute_smallest_enclosing_rectangle/1`, `compute_max_overall_distance/1`, `compute_convex_hull/1`, etc.
- **polygon**-related operations are available in `polygon.erl`:
  - **generation** of polygons: `get_triangle/3`, `get_upright_square/2`, `get_polygon/1`, etc.
  - **operations** on them: `get_diameter/1`, `get_smallest_enclosing_rectangle/1`, `get_area/1`, `is_in_clockwise_order/1`, `is_convex/1`, `to_string/1`, etc.
  - **rendering** them: `render/2`, `set_edge_color/2`, `get_edge_color/1`, `set_fill_color/2`, `get_fill_color/1`, etc.
  - managing their **bounding boxes**: `update_bounding_box/2`, etc.
- **bounding-boxes in general** are supported in `bounding_box.erl`, including `get_lazy_circle_box/1`, `get_minimal_enclosing_circle_box/1`, etc.

- a minimalist Runge-Kutta solver is defined in `rk4_solver.erl`

Data-Management Services

Some generic **data-structures**, in addition to the ones provided built-in with Erlang, are defined in `myriad/src/data-management`, notably:

- a set of **associative tables**, with a rather complete interface (to create, update, enrich, search, query, list, map, fold, merge, display, etc.) and various implementations thereof, tests and benchmarks, in:

  `{hash,lazy_hash,list_,tracked_hash,map_hash}table.erl`

- a `table` **pseudo-module** to abstract them out from the user's point of view; note that this is a fully virtual module, in the sense that neither `table.erl` nor `table.beam` exist (the `Myriad` parse transform replaces a call to the `table` module by, currently, a call to the `map_table` module; so, in order to consult the `table` API, please refer to `map_table.erl`)
- a way of **generating a read-only associative table** whose key/value pairs can be read from any number (potentially extremely large) of readers very efficiently (`const_table.erl`)
- a specific support for **other datatypes** (`pair.erl`, `option_list.erl`, `preferences.erl`, `tree.erl`)
- a first-level, optional support of the HDF5 file format (based on, and thus requiring, the enhanced fork we made of erlhdf5)

Finally, the `void/0` and `maybe/1` types are supported (thanks to the Myriad parse-transform).

Support for Code Injection

It may be useful to decide, at compile-time, based on tokens specified on the command-line, whether some code should be enabled.

Defining a token A *token* (a symbol) may or may not defined.

To define `my_token`, simply ensure that the a `-Dmy_token` command-line option is specified to the compiler (ex: refer to `ERLANG_COMPILER_TOKEN_OPT`, in `GNUmakevars.inc`).

To define `my_token` and set it to the integer value `127`, use the `-Dmy_token=127` command-line option. Values can also be floats (ex: `-Dmy_token=3.14`) or atoms (ex: `-Dmy_token=some_atom`).

A special token is `debug_mode`; if it is defined at all (and possibly associated to any value), the debug mode of Myriad is enabled.

Using tokens to enable code injection Various primitives for *code injection* are available in the `cond_utils` (mostly pseudo-) module.

`if_debug/1`, for example used as:

```
cond_utils:if_debug([A=B+1,io:format("Hello ~p",[A])])
```

will enable the specified code (either an arbitrarily nested expression or a list thereof) iff (if and only if) the `debug_mode` token has been defined.

Similarly, `if_defined/2`, for example used as:

```
cond_utils:if_defined(my_token,[EXPR1,EXPR2,...])
```

will inject `EXPR1`, `EXPR2`, etc. if `my_token` has been defined (any value associated to this token value will be ignored), otherwise the `if_defined/2` call will be removed as a whole[8].

As for `if_defined/3`, it supports two lists of expressions:

```
cond_utils:if_defined(a_token,FIRST_EXPR_LIST,SECOND_EXPR_LIST])
```

If `a_token` has been defined, the first list will be injected, otherwise the second will be.

Finally, with `if_set_to/{3,4}`, the injection will depend not only of a token being defined or not, but also onto the value (if any) to which it is associated.

An example with `if_set_to/3`:

```
cond_utils:if_set_to(some_token,42,EXPR_LIST)
```

will inject `EXPR_LIST` iff `some_token` has been defined and set to `42` (i.e. `-Dsome_token=42`). As a result, the specified expressions will not be injected if `some_token` has been set to another value, or not been defined at all. As usual, instead of a list of expressions, a single expression may be specified.

As for `if_set_to/4`, in:

[8] So `if_debug([...])` behaves exactly as: `if_defined(debug_mode,[...])`.

```
cond_utils:if_set_to(a_token,a_symbol,FIRST_EXPR_LIST,SECOND_EXPR_LIST)
```

`FIRST_EXPR_LIST` will be injected iff `a_token` has been defined and set to `a_symbol`, otherwise `SECOND_EXPR_LIST` will be.

Controlling assertions It may be convenient that, depending on a compile-time token (ex: in debug mode, typically triggered thanks to the `-Ddebug_mode` compilation option), *assertions* (expressions expected to evaluate to the atom `true`) are enabled, whereas they shall be dismissed as a whole should that token not be defined.

To define an assertion enabled in debug mode, use `assert/1`, like in:

```
cond_utils:assert(foo(A,B)=:=10)
```

Should at runtime the expression specified to `assert/1` be evaluated to a value `V` that is different from the atom `true`, a `{assertion_failed,V}` exception will be thrown.

More generally, an assertion may be enabled by any token (not `debug_mode` only) being defined, like in:

```
cond_utils:assert(my_token,bar(C))
```

Finally, an assertion may be enabled iff a token (here, `some_token`) has been defined and set to a given value (here, `42`), like in:

```
cond_utils:assert(some_token,42,not baz() andalso A)
```

This may be useful for example to control, on a per-theme basis, the level of checking performed, like in:

```
cond_utils:assert(debug_gui,1,basic_testing()),
cond_utils:assert(debug_gui,2,more_involved_testing()),
cond_utils:assert(debug_gui,3,paranoid_testing()),
```

Note that, in this case, a given level of checking should include the one just below it (ex: `more_involved_testing()` should call `basic_testing()`).

For more information Refer for usage and stubs to the `cond_utils` module (defined in `myriad/src/meta`), knowing that it is actually implemented thanks to the Myriad parse transform.

For examples and testing, see the `cond_utils_test` module, available at the same location.

Helpers For User Interface Programming

Some services have been defined, in `myriad/src/user-interface`, in order to handle more easily interactions with the user, i.e. to provide a user interface.

> **Note**
>
> The user-interface services, as a whole, are currently *not* functional. A rewriting thereof as been started yet has not completed yet.

Various Flavours of User Interfaces Such a user interface may be:

- either **text-only**, within a console, relying either on the very basic `text_ui` (for raw text) or its more advanced `term_ui` counterpart (for terminal-based outputs)
- or **graphical**, with `gui`

Text-based user interfaces are quite useful, as they are lightweight, incur few dependencies (if any), and can be used with headless remote servers (`text_ui` and `term_ui` work well through SSH, and require no X server nor mouse).

As for graphical-based user interfaces, they are the richest, most usual, and generally the most convenient, user-friendly interfaces.

The user interfaces provided by Myriad are stateful, they rely on a **state** that can be:

- either `explicit`, in a functional way; thus having to be carried in all calls
- or `implicit`, using - for that very specific need only - the process dictionary (even if we try to stay away of it as much as possible)

We tested the two approaches and preferred the latter (implicit) one, which was found considerably more flexible and thus finally fully superseded the (former) explicit one.

We made our best so that a lower-level API interface (relying on a more basic backend) is **strictly included** in the higher-level ones (ex: `term_ui` adds concepts - like the one of window or box - to the line-based `text_ui`), in order that any program using a given user interface may use any of the next, upper ones as well (provided implicit states are used), in the following order: the `text_ui` API is included in the one of `term_ui`, which is itself included in the one of `gui`.

We also defined the **settings table**, which is a table gathering all the settings specified by the developer, which the current user interface does its best to accommodate.

Thanks to these "Matryoshka" APIs and the settings table, the definition of a more generic `ui` interface has been possible. It selects automatically, based on available local software dependencies, **the most advanced available backend**, with the most relevant settings.

For example a relevant backend will be automatically selected by:

```
$ cd test/user-interface/src
$ make ui_run
```

On the other hand, if wanting to select a specified backend:

```
$ make ui_run CMD_LINE_OPT="--use-ui-backend term_ui"
```

(see the corresponding GNUmakefile for more information)

Raw Text User Interface: `text_ui` This is the most basic, line-based monochrome textual interface, directly in raw text with no cursor control.

Located in `{src,test}/user-interface/textual`, see `text_ui.erl` for its implementation, and `text_ui_test.erl` for an example of its use.

Terminal Text User Interface: `term_ui` This is a more advanced textual interface than the previous one, with colors, dialog boxes, support of locales, etc., based on dialog (possibly whiptail could be supported as well). Such backend of course must be available on the execution host then.

For example, to secure these prerequisites:

```
# On Arch Linux:
$ pacman -S dialog

# On Debian-like distros:
$ apt-get install dialog
```

Located in `{src,test}/user-interface/textual`, see `term_ui.erl` for its implementation, and `term_ui_test.erl` for an example of its use.

Graphical User Interface: `gui` This interface relied initially on `gs` (now deprecated), now on wx (a port of wxWidgets), maybe later in HTML 5 (possibly relying on the Nitrogen web framework for that). For the base dialogs, Zenity could have been on option.

> **Note**
> GUI services are currently being reworked, to provide a `gs`-like concurrent API while relying underneath on `wx`, with some additions (such as canvases).

The goal is to provide a small, lightweight API (including message types) that are higher-level than `wx`, and do not depend on any particular GUI backend (such as `wx`, `gs`, etc.) to avoid that user programs become obsolete too quickly, as backends for GUI rise and fall relatively often.

So for example the messages received by the user programs shall not mention `wx`, and they should take a form compliant with WOOPER message conventions, to easily enable user code to rely on WOOPER if wanted.

Located in `{src,test}/user-interface/graphical`, see `gui.erl`, `gui_color.erl`, `gui_text.erl`, `gui_canvas.erl`, etc., with a few tests (`gui_test.erl`, `lorenz_test.erl`).

Related information of interest:

- wxErlang: Getting started and Speeding up, by Arif Ishaq
- http://wxerlang.dougedmunds.com/

All-Purpose Helper Scripts

A small set of scripts has been defined, in `myriad/src/scripts`, in order to help:

- finding in (Erlang) source code **type definitions** (`find-type-definition.sh`, `find-record-definition.sh`) and **function specifications** (`find-function-specification.sh`
- **benchmarking** Erlang code: `benchmark-command.escript`, `benchmark-command.sh`, `etop.sh`
- generating **documentation**: `generate-docutils.sh`, `generate-pdf-from-rst.sh`
- supporting **explicit typing**: `list-available-types.sh`, `add-deduced-type-specs.escript`
- evaluating Erlang **code size**: `make-code-stats.sh`
- **running** Erlang programs: `launch-erl.sh`, i.e. the (non-trivial) script that is automatically called by all our execution rules (i.e. we always run our Erlang programs through it)
- parsing **XML** thanks to `xmerl`: `show-xml-file.escript`

To be added: merging facilities (upcoming `merge-tree.escript`)

Utility Toolbox

This is the **core** of the Ceylan-Myriad library: a toolbox comprising many helper functions (with their tests), defined in the myriad/src/utils directory, often providing enhanced, more specialised services compared to the ones offered by the Erlang standard libraries.

These helpers (code and typing information) are thematically aggregated in modules that are generally suffixed by _utils, and include:

- many **basic, general-purpose services**, defined in basic_utils.erl, regarding:
  - the base types we defined
  - process registration
  - notifications
  - message handling
  - many miscellaneous functions
- **cipher**-related facilities (basic, a bit exotic chained symmetric encryptions, notably with Mealy machines), in cipher_utils.erl
- functions to manage Erlang **compiled BEAM code** (code_utils.erl)
- services to manage the **execution of other programs** (executable_utils.erl), to:
  - locate said executables
  - to execute functional services (ex: display a PDF) regardless of the actual executable involved
- services to handle more easily the (UNIX) shells and also the command-line arguments (a bit like getopt), regardless of the interpreter or escript context (shell_utils.erl)
- helpers for **file-based** I/O operations (file_utils.erl)
- a very basic support of **Finite State Machines** (fsm_utils.{e,h}rl)
- a few operations defined on **graphs** (graph_utils.erl, with find_breadth_first/{3,4})
- extra operations defined on **lists** (list_utils.erl), including rings
- support for **network**-related concerns (net_utils.erl.{e,h}rl)
- services to offer **randomness** (random.erl), with regard to various sources (the Erlang built-in algorithm, crypto, newer ones like exsplus - our current default, exs64 and exs1024), for seeding, drawing, etc.
- very little support of **RDF** operations, standing for Resource Description Framework (rdf_utils.erl)
- facilities to perform **REST calls** (rest_utils.erl), using built-in httpc and http_client, and possibly a JSON parser, jsx

- elements for the sending of **SMS** (`sms_utils.erl`), based either on third-party providers providing REST APIs, or via a mobile phone (typically connected thanks to a USB cable)
- support for operations at the **operating-system** level (`system_utils.{e,h}rl`)
- services to handle **text** (`text_utils.erl`)
- functions to manage **time** information (`time_utils.erl`)
- a few helpers to ease the writing of escripts relying on the Myriad layer (`script_utils.erl`)
- services about all kinds of **units** (`unit_utils.erl`); refer to the Management of Units section below for more information
- very basic facilities for **applications**, in `app_facilities.{e,h}rl` with an example (`most_basic_example_app.erl`)

Support for Metaprogramming

Over time, quite a lot of developments grew to form primitives that manage ASTs (*Astract Syntax Trees*), based on Erlang's parse transforms.

These developments are gathered in the `src/meta` directory, providing notably:

- `meta_utils.{e,h}rl`: basic primitives to **transform ASTs**, with a bit of testing (`meta_utils_test`)
- `type_utils`: a still rather basic toolbox to **manage data types** - whether built-in, compound or parametrised (expressed as strings, as terms, etc.)
- `ast_*` modules to handle the various **elements that can be found in an AST** (ex: `ast_expression`, `ast_type`, `ast_pattern`, etc.)

Finally, a few usage examples of these facilities are:

- `minimal_parse_transform_test`: the simplest parse transform test that we use, typically operating on `simple_parse_transform_target`
- `example_parse_transform`: a rather minimal parse transform
- `myriad_parse_transform`: the parse transform used within `Myriad`, transforming each and every module of that layer (and of at least some modules of upper layers)

So the purpose of this parse transform is to **convert ASTs that are Myriad-compliant into ASTs that are directly Erlang compliant**.

For that, following changes are operated:

- in type specifications, the Myriad-specific `void/0`, `maybe/1` types are adequately translated:
  - `void()` becomes `basic_utils:void()`, a type alias of `any()`, made to denote returned terms that are not expected to be used by the caller (as if that function's only purpose was its side-effects)
  - `maybe(T)` becomes the type union `'undefined'|T`
- both in type specifications and actual code, `table/2`, the Myriad-specific associative table pseudo-type, is translated into an actual table type:
  - by default, `map_hashtable` (the generally most efficient one)
  - unless it is overridden on a per-module basis with the `table_type` define, like in: `-table_type(list_table).`
- the `cond_utils` services will drive conditional code injection

More generally, Myriad offers the support to traverse *any* AST (the whole Erlang grammar is supported, in its abstract form) and to **transform** it (ex: an expression being removed, transformed or replaced by other expressions), with the ability for the user to define his own type/call replacement mappings, or more general transformation functions to be triggered when specified elements are found in the AST (ex: remote calls with relevant MFA).

The traversal may be done in a stateful manner, i.e. any user-defined transformation will be able to access (read/write) any state of its own in the course of the traversal.

As a result, a single pass through the input AST may be done, in which any kind of transformations may be applied, resulting in another (Erlang-compliant) AST being output and afterwards compiled.

Management of Units

Motivation A value of a given type (ex: a float) can actually correspond to quantities as different as meters and kilowatts per hour.

Therefore **units shall preferably be specified alongside with values being processed**, and a language to express, check and convert these units must be retained. Of course units are of interest as other metadata are - such as accuracy, semantics, etc.

Available Support The `Myriad` layer provides such a service, in a very basic, ad hoc form (which is useful to introduce "special" non-physical, non-standard units, such as `euro/year`), meant to be enriched over time.

Specifying Units

Aliases For convenience, *aliases* of units can be defined, i.e. alternate names for a given canonical unit. For example the Hertz unit (`Hz`) is an alias of the `s^-1` (per-second) canonical unit.

Built-in Units So one may use the following **built-in units**, whose symbol[9] is specified here between brackets, like in `"[N.m]"` (an alternate notation is to prefix a unit with `U:`, like in `"U: N.m"`):

- the seven SI base units, namely:
  - meter, for length [`m`]
  - gram, for mass [`g`][10] (note: this is a footnote, not an exponent!)
  - second, for time [`s`]
  - ampere, for electric current [`A`]
  - kelvin, for thermodynamic temperature [`K`]
  - mole, for the amount of substance [`mol`]
  - candela, for luminous intensity [`cd`]
- the usual derived units, notably:
  - hertz, for frequency [`Hz`]
  - degree, for degree of arc [`°`] (not supported yet)
  - radian, for angle [`rad`] (not supported yet)
  - steradian, for solid angle [`sr`] (not supported yet)
  - newton, for force, weight [`N`]
  - pascal, for pressure, stress [`Pa`]
  - joule, for energy, work, heat [`J`]
  - watt, for power, radiant flux [`W`]
  - coulomb, for electric charge, quantity of electricity [`C`]
  - volt, for voltage, electrical potential difference, electromotive force [`V`]

  - farad, for electrical capacitance [`F`]
  - ohm, for electrical resistance, impedance, reactance [`Ohm`]
  - siemens, for electrical conductance [`S`]
  - weber, for magnetic flux [`Wb`]
  - tesla, for magnetic field strength, magnetic flux density [`T`]
  - henry, for inductance [`H`]
  - lumen, for luminous flux [`lm`]
  - lux, for illuminance [`lx`]
  - becquerel, for radioactive decays per unit time [`Bq`]
  - gray, for absorbed dose of ionizing radiation [`Gy`]
  - sievert, for equivalent dose of ionizing radiation [`Sv`]
  - katal, for catalytic activity [`kat`]

- the units **widely used** in conjunction with SI units (note that they may not respect the principle of being a product of integer powers of one or more of the base units):

  - litre, for $10^{-3}m^3$volumes [`L`]
  - tonne, for 1,000 kilogram masses [`t`]
  - electronvolt, for 1.602176565(35).10-19 joule energies [`eV`]
  - minute, for 60-second durations [`min`]
  - hour, for 60-minute durations [`h`]
  - day, for 24-hour durations [`day`]
  - week, for 7-day durations [`week`]

- the **special** units (they generally cannot map directly to any SI unit, yet can be handled separately), designating:

  - month [`month`] (correspondence to base time units unspecified, as this duration is not constant; ex: a month can be 29, 30 or 31 days)
  - year [`year`] (correspondence to base time units unspecified, as this duration is not constant; ex: a year can be 365, 366 or 365.25 days, etc.)
  - degree Celsius, for temperature relative to 273.15 K [`°C`] (see note below)
  - dimension-less quantities (ex: an index) [`dimensionless`] (most probably clearer than `m/m`)
  - a count, i.e. a dimensionless number, generally a positive integer [`count`] (ex: `14`), considered as an alias of `dimensionless`
  - a ratio, i.e. a dimensionless floating-point value, generally displayed as a percentage [`ratio`] (ex: `-12.9%`); another alias of `dimensionless`
  - currencies, either [`$`] (US Dollar) or [`euros`] (Euro), whose exchange rates of course vary

- values whose unit has not been specified [unspecified_unit]

- metric prefixes thereof, i.e. multiples and sub-multiples of the units previously mentioned; currently the supported prefixes are:
  - yotta, i.e. 10^{24} [Y]
  - zetta, i.e. 10^{21} [Z]
  - exa, i.e. 10^{18} [E]
  - peta, i.e. 10^{15} [P]
  - tera, i.e. 10^{12} [T]
  - giga, i.e. 10^{9} [G]
  - mega, i.e. 10^{6} [M]
  - kilo, i.e. 10^{3} [k]
  - hecto, i.e. 10^{2} [h]
  - deca, i.e. 10 [da]
  - deci, i.e. 10^{-1} [d]
  - centi, i.e. 10^{-2} [c]
  - milli, i.e. 10^{-3} [m]
  - micro, i.e. 10^{-6} [µ]
  - nano, i.e. 10^{-9} [n]
  - pico, i.e. 10^{-12} [p]
  - femto, i.e. 10^{-15} [f]
  - atto, i.e. 10^{-18} [a]
  - zepto, i.e. 10^{-21} [z]
  - yocto, i.e. 10^{-24} [y]

Note

There is a problem with temperatures, as they can be expressed at least in kelvins or degrees Celsius, whereas the two corresponding scales do not match, since there is an offset:

[K] = [°C] + 273.15

As a result, unit conversions would require updating as well the corresponding value, and, more generally, they should be treated as fully distinct units (ex: `kW/°C` cannot be automatically converted in terms of SI base units, i.e. using `K`).
This is why we "degraded" Celsius degrees, from a derived unit to a special one.
The same applies to the Fahrenheit unit (a likely addition), as:

[°C] = 5/9.([°F]-32)

[9] To avoid requesting the user to type specific Unicode characters, we transliterated some of the symbols. For example, instead of using the capital Omega letter, we used `Ohm`.

[10] We preferred here deviating a bit from the SI system, by using this non-prefixed unit (the *gram*) instead of the SI standard one, which happens to be the *kilogram*.

Composing One's Units So an actual unit can be composed from the aforementioned built-in units (be they base, derived, widely used, special units; prefixed or not)[11], using two built-in operators, which are `"."` (multiply, represented by the dot character - not `"*"`) and `"/"` (divide, represented by the forward slash character).

The resulting type shall be specified as a string, containing a series of built-in units (potentially prefixed) alternating with built-in operators, like in: `"kW.s/m"`.

> **Note**
> As a result, `"kWh"` is not a valid unit: it should be denoted as `"kW.h"`. Similarly, `"W/(m.k)"` is not valid, since parentheses are currently not supported: `"W/m/k"` may be used instead.

Finally, exponents can be used as a shorthand for both operators (ex: `kg.m^2.s^-1`, instead of `kg.m.m/s`). They should be specified explicitly, thanks to the caret character (`"^"`); for example `"m^2/s"`, not `"m²/s"`.

If deemed both safe and useful, we may consider in the future performing:

- symbolic unit checking (i.e. determining that a derived unit such as `N.s` (newton.second) is actually, in canonical SI units, `m^2.kg.s^-1`), and thus that values of these two types can safely be used indifferently in computations
- automatic value conversions (ex: converting `km/hour` into `m/s`), provided that the overall computational precision is not significantly deteriorated

The corresponding mechanisms (type information, conversion functions, unit checking and transformation, etc.) are defined in `unit_utils.erl` and tested in `unit_utils_test.erl`, in the `myriad/src/utils` directory.

Checking Units A typical example:

```
1> MyInputValue="-24 mS.m^-1".
2> {Value,Unit}=unit_utils:parse_value_with_unit(MyInputValue).
3> io:format("Corresponding value: ~f.~n", [ Value ] ).
Corresponding value: -24.0.
4> io:format("Corresponding unit: ~s.~n",
   [unit_utils:unit_to_string(Unit)]).
"s^3.A^2.g^-1.m^-3, of order -6"
5> unit_utils:value_with_unit_to_string(Value,Unit).
"-2.4e-5 s^3.A^2.g^-1.m^-3"
```

Possible Improvements Regarding Dimensional Analysis Some programming languages provide systems to manage dimensional information (ex: for physical quantities), generally through add-ons or libraries (rarely as a built-in feature).

A first level of support is to provide, like here, an API to manage units. Other levels can be:

1. to integrate unit management directly, seamlessly in language expressions, as if it was built-in (as opposed to having to use explicitly a third-party API for that); for example at least half a dozen different libraries provide that in Python

[11] In the future, defining an actual unit from other actual units might be contemplated.

2. to be able to define "polymorphic units and functions", for example to declare in general that a speed is a distance divided by a duration, regardless of the possible units used for that

3. to perfom *static* dimensional analysis, instead of checking units at runtime

The two latter use cases can for example be at least partially covered by Haskell libraries.

SQL support

About SQL Some amount of SQL (*Structured Query Language*) support for relational database operations is provided by the `Myriad` layer.

As this support relies on an optional prerequisite, this service is disabled by default.

Database Back-end To perform SQL operations, a corresponding software solution must be available.

The SQL back-end chosen here is the SQLite 3 library. It provides a self-contained, serverless, zero-configuration, transactional SQL database. It is an embedded SQL database engine, as opposed to server-based ones, like PostgreSQL or MariaDB.

It can be installed on Debian thanks to the `sqlite3` and `sqlite3-dev` packages, `sqlite` on Arch Linux..

We require version 3.6.1 or higher (preferably: latest stable one). It can be checked thanks to `sqlite3 --version`.

Various related tools are very convenient in order to interact with a SQLite database, including `sqlitebrowser` and `sqliteman`.

On Arch Linux, one can thus use: `pacman -Sy sqlite sqlitebrowser sqliteman`.

Testing the back-end:

```
$ sqlite3 my_test
```

```
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> create table tblone(one varchar(10), two smallint);
sqlite> insert into tblone values('helloworld',20);
sqlite> insert into tblone values('my_myriad', 30);
sqlite> select * from tblone;
helloworld|20
my_myriad|30
sqlite> .quit
```

A file `my_test`, identified as `SQLite 3.x database`, must have been created, and can be safely removed.

Erlang SQL Binding This database system is directly accessed thanks to an Erlang binding.

Two of them have been identified as good candidates:

- erlang-sqlite3: seems popular, with many contributors and users, actively maintained, based on a `gen_server` interacting with a C-node, involving only a few source files
- esqlite: based on a NIF, so more able to jeopardize the stability of the VM, yet potentially more efficient

Both are free software.

We finally preferred `erlang-sqlite3`.

By default we consider that this back-end has been installed in `~/Software/erlang-sqlite3`. The `SQLITE3_BASE` variable in `myriad/GNUmakevars.inc` can be set to match any other install path.

Recommended installation process:

```
$ mkdir ~/Software
$ cd ~/Software
$ git clone https://github.com/alexeyr/erlang-sqlite3.git
Cloning into 'erlang-sqlite3'...
remote: Counting objects: 1786, done.
remote: Total 1786 (delta 0), reused 0 (delta 0), pack-reused 1786
Receiving objects: 100% (1786/1786), 3.24 MiB | 570.00 KiB/s, done.
Resolving deltas: 100% (865/865), done.
Checking connectivity... done.
$ cd erlang-sqlite3/
$ make
rm -rf deps ebin priv/*.so doc/* .eunit/* c_src/*.o config.tmp
rm -f config.tmp
echo "normal" > config.tmp
./rebar get-deps compile
==> erlang-sqlite3 (get-deps)
==> erlang-sqlite3 (compile)
Compiled src/sqlite3_lib.erl
Compiled src/sqlite3.erl
Compiling c_src/sqlite3_drv.c
[...]
```

Testing the binding:

```
make test
./rebar get-deps compile eunit
==> erlang-sqlite3 (get-deps)
==> erlang-sqlite3 (compile)
==> erlang-sqlite3 (eunit)
Compiled src/sqlite3.erl
Compiled src/sqlite3_lib.erl
Compiled test/sqlite3_test.erl
======================== EUnit ========================
module 'sqlite3_test'
  sqlite3_test: all_test_ (basic_functionality)...[0.002 s] ok
  sqlite3_test: all_test_ (table_info)...ok
  [...]
  sqlite3_lib: delete_sql_test...ok
  sqlite3_lib: drop_table_sql_test...ok
  [done in 0.024 s]
  module 'sqlite3'
=======================================================
All 30 tests passed.
Cover analysis: ~/Software/erlang-sqlite3/.eunit/index.html
```

Pretty reassuring.

SQL Support Provided By the *Myriad* Layer To enable this support, once the corresponding back-end (see Database Back-end) and binding (see Erlang SQL Binding) have been installed, the `USE_SQLITE` variable should be set to `true` in `myriad/GNUmakevars.inc` and `Myriad` shall be rebuilt.

Then the corresponding implementation (`sql_support.erl`) and test (`sql_support_test.erl`), both in `myriad/src/data-management`, will be built (use `make clean all` from the root of `Myriad`) and able to be run (execute `make sql_support_run` for that).

Testing it:

```
$ cd myriad/src/data-management
$ make sql_support_run
       Compiling module sql_support.erl
       Compiling module sql_support_test.erl
       Running unitary test sql_support_run
 [...]
 --> Testing module sql_support_test.
 Starting SQL support (based on SQLite3).
 [...]
 Closing database.
 Stopping SQL support.
 --> Successful end of test.
 (test finished, interpreter halted)
```

Looks good.

SQL-related Troubleshooting

Compiling module sql_support.erl : can't find include file "sqlite3.hrl"

- `USE_SQLITE` not set to `true` in `myriad/GNUmakevars.inc`
- `erlang-sqlite3` back-end not correctly installed (ex: `SQLITE3_BASE` not pointing to a right path in `myriad/GNUmakevars.inc`)

`Myriad` Main Conventions

Execution Targets Two execution target modes have been defined:

- `development` (the default): meant to simplify the task of developers and maintainers by reporting as much information and context as possible, even at the expense of some performances and reliability (ex: no retry in case of failure, shorter time-outs not to wait too long in case of errors, more checks, etc.)
- `production`: mostly the reciprocal of the `development` mode, whose purpose is to favor efficient, bullet-proof operations

These execution targets are *compile-time* modes, i.e. they are set once for all when building the layer at hand (probably based, if using OTP, on the rebar corresponding modes - respectively `dev` and `prod`).

See `EXECUTION_TARGET` in `GNUmakevars.inc` to read and/or set them.

The current execution target is of course available at runtime on a per-layer level, see `basic_utils:get_execution_target/0` for more information.

This function shall be compiled once per layer to be accurate, in one of its modules. It is just a matter of adding the following include in such module:

```
-include_lib("myriad/utils/basic_utils.hrl").
```

Other Conventions

- for clarity, we tend to use longer variable names, in CamelCase
- we tend to use mute variables to clarify meanings and intents, as in `_Acc=[]` (beware, despite being muted, any variable in scope that bears the same name will be matched)
- as there is much list-based recursion, a variable named `T` means `Tail` (as in `[Head|Tail]`)

See also the few hints regarding contribution.

`Myriad` Gotchas

Header dependencies Only a very basic dependency between header files (`*.hrl`) and implementation files (`*.erl`) is managed.

As expected, if `X.hrl` changed, `X.beam` will be recompiled whether or not `X.erl` changed. However, any `Y.erl` that would include `X.hrl` would not be automatically recompiled.

Typically, when in doubt after having modified a record in a header file, just run `make rebuild` from the root of that layer (build is fast anyway, as quite parallel).

About the `table` module This is a pseudo module, which is not meant to exist as such (no `table.erl`, no `table.beam`).

The `Myriad` parse transform replaces references to the `table` module by references to the `map_hashtable` module. See table transformations for more information.

Support for `Myriad`

So you respected the prerequisites and build sections, and something went wrong? Generally we made sure that any detected error is blocking and loudly reported, with as much context as possible.

The simpler solution is then to create a relevant issue.

For all other needs, please drop an email to the address listed on top of document. We do our best to answer on a timely basis.

Finally, provided that they meet licensing terms, scope and quality standards, contributions of all sorts are vey welcome, be them porting efforts, increased test coverage, functional enrichments, documentation improvements, code enhancements, etc. See the next section for additional guidelines.

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Contributions & Ending Word

Each time that you need a basic service that:

- seems neither provided by the Erlang built-in modules nor by this `Myriad` layer
- is generic-enough, simple and requires no special prerequisite

please either enrich our `*_utils.erl` helpers, or add new general services!
In such a case, we would prefer that, in contributed code:

- Myriad code style is, as much as possible, respected (regarding naming, spacing, code/comments/blank line ratios, etc.)
- lines stop no later than their 80th character
- whitespaces be removed (ex: one may use the `whitespace.el` Emacs mode)

Thanks in advance, and have fun with Myriad!