Technical Manual of the Ceylan-Myriad Layer

Organisation: Copyright (C) 2008-2022 Olivier Boudeville

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

Creation date: Sunday, August 17, 2008

Lastly updated: Monday, October 17, 2022

Version: 1.0.27

Status: Stable

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 either to browse the Myriad API documentation or simply to read the corresponding source files, which are intensely commented and generally straightforward.

The latest version of this documentation is to be found at the official Ceylan-Myriad website (`http://myriad.esperide.org`).

This documentation is also mirrored here.

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 yet organised, uniform, consistent **code library**), in their most convenient, reliable, efficient version, together with their specification, documentation and testing.

This **toolbox layer** provides its (generally lightweight, simple) services just on top of the Erlang[3] language, as a relatively small (comprising currently about 115k lines of code, including blank ones and comments), thin layer. Supported platforms are most Unices (including of course GNU/linux) and also - but to a lesser extent, i.e. not tested on a daily basis - Windows.

These services tend to stay away from introducing any new **dependency**; however many supports (of OpenGL, JSON, Protobuf, SQL, HDF5, etc.; also in terms of GUI or integration with other languages) may be useful, and are to rely on well-chosen and well-integrated backends.

So, 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** (see the Third-Party Dependencies section for more details), in a logic of containment: if, for any reason, a dependency has to be replaced, the goal is that the user code can remain, as much as realistically possible, happily unaware of it - only the Myriad integration layer being impacted.

The purpose of Myriad is not to fully integrate the many backends it *may* rely on (it would be an unachievable task) but, once a proper integration base has been defined for a given backend, to pragmatically increase its coverage based on the actual needs.

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` layer.

The project repository is located here. We gathered also some more general Erlang-information in our corresponding HOWTO.

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

[3] If needing to discover/learn Erlang, we recommend browsing Learn You Some Erlang for great good! or, even better, buying their book!
See also our Erlang HOW-TO.

- 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.

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

Supported Platforms

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

Myriad can be built and tested successfully on the `Windows` platform; for that we rely on MSYS2. Refer to this section of Ceylan-Hull for information regarding our recommended Windows developer settings in terms of shells and general environment.

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

Software Prerequisites

The main tool prerequisite is of course having the Erlang environment available, in its `24.0` version[5] or more recent.

There are various ways of obtaining it (from your distribution[6], 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
```

As using a Just-In-Time compiler increases the performances significantly, we chose to force its use. As a result, a C++ 17 compiler, like a recent enough `g++` one, is required by our script.

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.). See also the Base GUI Backend section to secure any related prerequisite.

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.

---

[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 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.

To determine programmatically the recommended version of Myriad-based code, just execute our install-erlang.sh script with its `--version` option (this is the sole source of reference, and is used in our full software stack).

[6] For example, as root: `apt-get install erlang` on Debian, `pacman -Sy erlang` in Arch Linux, etc.; just ensure that its version is not outdated then. Refer to our Erlang HOWTO for further details.

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.

Note that by default our native, make-based, build system is used (see the build section of Ceylan-HOWTOs for further details). Alternatively, a rebar3-based build can be done (refer to the OTP Build section).

In this case 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 - this make target is useful for the layers built on top of Myriad).

So, alternatively to using `make` directly, one may execute `rebar3 compile` instead.

Testing Myriad

As Myriad has no prerequisite (besides Erlang itself of course), just run (possibly simply thanks to `rebar3 compile` after a `git clone https://github.com/Olivier-Boudeville/Ceylan-M`
still from the root directory of Myriad:

```
$ make test
```

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

> **Note**
>
> Myriad is built and tested at each commit through continuous integration. The same holds for the projects based on it, directly (ex: WOOPER, Seaplus) or not (ex: Traces, Mobile, US-Web), so in terms of usability, confidence should be high.

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
```

Maintaining Myriad and Deriving Projects with regard to rebar3

For Myriad as for all developments built on top of it (ex: specialisation layers or applications), any dependency may be specified in their `rebar.config`[7] through a branch of a GIT repository corresponding to that dependency.

For example, Myriad itself does not require any specific dependency, but projects making use of Myriad (ex: WOOPER) may specify in their `rebar.config`:

```
{deps, [{myriad, {git, "git://github.com/Olivier-Boudeville/Ceylan-Myriad",
                                        {branch, "master"}}}]}.
```

However, when having to build a dependency, rebar3 will not refer to the tip of the branch specified for it, but to any commit it may read from any pre-existing `rebar.lock` file at the root of the current project (the underlying goal being to allow for more reproducible builds).

As the rebar3 recommandation is to store a version of that lock file in source version control, **it shall be regularly updated** otherwise the dependencies of a given project will stick, for the worst, to an older version of their branch, designated by an obsolete reference (this can be detected for example when continuous integration breaks after a nevertheless legit commit of the project).

The solution is thus, for a project of interest, to regularly force an update of its dependencies referenced in its own lock file, and to commit the resulting version.

For example, one would issue from the root of the project of interest:

```
$ rebar3 upgrade
```

This may update the `ref` entry of its dependencies (including Myriad) in its `rebar.lock` file, which shall then be committed for posterity.

OTP Build

These build considerations apply to Myriad but also, more generally, to most if not all our Erlang developments.

---

[7] For example, with the conventions we rely on, `rebar.config` is generated from the `conf/rebar.config.template` file of the project of interest.

Why Providing Two Different Build/Deploy/Run Systems

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, and has been fully satisfactory for years (simple, lightweight, reliable, controllable, flexible, fast, etc.).

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.

More precisely we routinely (see WOOPER or Seaplus) rely on layers built on top of Myriad, which define their own parse transforms that are themselves parse-transformed by Myriad's one - and it works great.

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.

So currently all our Erlang-based developments can also be built and tested through rebar3, and this support is checked at each commit thanks to continuous integration.

We use less frequently releases (we rely on a basic deployment procedure of our own) and even less hex, yet they were supported once, so we believe that their integration should be at least fairly close to be operational (if not, patches welcome!).

Relying on Rebar3

Despite the kind support of the rebar3 authors and much time spent on its integration, sometimes our build based on it (for Myriad and the projects built on top of it) has encountered issues or has been lagging behind our native one.

Now we believe that all pending issues have been solved (rebar3 is a neat tool), yet being able to switch back to another lighter, ad-hoc, more controlled build system is sometimes a relief - at least a welcome security. Anyway the user can choose between these two (native vs rebar3) build machineries. As for us, we still prefer our native build system, even if it leaves to the developer the task of installing the needed prerequisites by him/herself.

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[8].

[8] 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.

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
```

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 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[9].

A full, autonomous, functional by design build procedure can be also found in Myriad's continuous integration script.

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
```

[9]The rebar-based build relies, thanks to {pre,post}-compile hooks, on our native build system. Because of extraneous, failing recompilations being nevertheless triggered by rebar, we had to introduce bullet-proof hooks (refer to 1, 2).

```
   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] [...]

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.

Note

Finally our workflow does not rely on Hex, so we do not update the Hex packages anymore. Just drop us an email if needing an updated one.

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.

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. Support for code injection
6. Helpers for graphical user interface (GUI) programming
7. All-purpose helper scripts
8. Spatial support
9. Utility toolbox
10. Management of units
11. Metaprogramming, based on heavy use of parse transforms
12. 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 either to browse the Myriad API documentation or simply 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 system, 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 some kinds of targets (ex: all source files)
  - `GNUmakerules-explicit.inc`, for all "direct" (explicitly designated) 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 Make files that remain rather minimal, as they mostly specify the relative base path and only refer to the generic variables and rules; see `myriad/src/GNUmakefile` as an example thereof

An example of this stacked build 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).

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

As an example, the `shell` make target allows to spawn an Erlang shell that is readily able to use the current layer and all the ones below (including thus Myriad); this way, direct, hassle-free interactive testing can be performed - which may prove quite convenient.

General Settings

These general-purpose settings and helpers, gathered in Myriad's conf directory, deal with:

- default CSS files (`Default-docutils.css`)
- our recommended versions of (commented) configuration files for various tools, notably `Emacs`: see our init.el, to be placed in the `~/.emacs.d/` directory; now our configuration is standalone (no need for extra files/packages) and cross-platform (at least Unices and Windows)
- 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 (refer to the Spatial Support section for more details); 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 surfaces**: `update_bounding_surface/2`, etc.
- **bounding spaces** (notably bounding boxes) are supported in `bounding_{surface,volume}.erl`, including `get_lazy_bounding_circle/1`, `get_minimal_enclosing_circle/1`, etc.

- a minimalist Runge-Kutta solver is defined in `rk4_solver.erl` (see also the corresponding Lorenz test)

Data-Management Services

Datatypes

Some generic **data-structures**, in addition to the ones provided built-in with Erlang, are defined in `myriad/src/data-management`, and described below.

Table Types A set of types for **associative tables** is available, each offering a rather complete interface (to create, update, enrich, search, query, list, map, fold, merge, display, etc. a table - or entries thereof) and a different trade-off.

Various implementations are defined (with tests and benchmarks), in:

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

A `table` **pseudo-module** is additionally provided, in order to abstract out these various options: the user may then rely exclusively on `table`, regardless of the actual table type this pseudo-module will be translated at compilation-time.

Indeed the `table` module is a fully virtual one, in the sense that neither `table.erl` nor `table.beam` exists, and that the `Myriad` parse transform is to automatically replace a call to this `table` pseudo-module by a call to one of the aforementioned table types (and it will do the same replacement in type specifications as well).

By default, such `table` calls will be translated to corresponding calls to our `map_hashtable` module, which is generally the most efficient one (it relies on the more recently-introduced Erlang `maps` built-in datatype, which `table` now favors).

As a result, in order to consult the `table` API, please refer to `map_hashtable.erl`.

Such a default implementation may be overridden on a per-module basis, thanks to a `table_type` define.

For example, specifying `-table_type(list_table).` will result in the current module to translate `table` to `list_table`, instead of the default `map_hashtable`.

Another type of table is the `bijective_table`, which allows efficient (runtime) bidirectional conversions between two sets, each having unique elements (no duplicates).

As a mere convention, when one set is dealing with internal identifiers and the other on third-party ones, we recommend that the internal identifiers are selected as the first elements, and the third party as second elements.

Finally, a way of **generating read-only associative tables** whose key/value pairs can be read very efficiently from any number (potentially extremely large) of readers (processes) is provided with `const_table.erl` (refer to `const_table_test.erl` for a test thereof).

No ETS table, replication (ex: per-process table copy) or message sending is involved: thanks to meta-programming, a module is generated on-the-fly, exporting as many functions as there are different keys in the table of interest. Calling a function corresponding to a key returns its associated value.

More precisely, a module name (ex: `foobar`) and a list of `{atom(), type_utils:permanent_term()}`[10] entries shall be provided to `const_table:generate_in_{memory,file}/2`; then, for each key/value pair in the specified table (ex: `{baz, 42.0}`), a 0-arity function is generated and exported in that module, as if we had:

```
-module(foobar).
```

```
[...]

-export([baz/0]).

-spec baz() -> term().
baz() ->
  42.0.
```

Then third-party code can call for example `foobar:foo()` and have `42.0` returned. This is presumably the most efficient way of sharing constants in Erlang between many processes (supposedly at least on par with persistent_term).

Note that with `generate_in_memory/2` no actual module file is created (ex: no `foobar.beam` file is ever written in the filesystem): the operation remains fully in-memory (RAM). With `generate_in_file/{2,3}` a suitable module file is written on disk, so that the corresponding module can be loaded in the future like any other module.

Keys must be atoms, and the table of interest shall be immutable (const), even if, thanks to hot code upgrade, one may imagine updating the table at will, having any number of successive versions of it.

Generating a table of the same name more than once should be done with care, as if a given table is generated thrice (hence updated twice), the initial table would first switch from "current" to "old", and then would be removed. Any process that would linger in a function of this module would then be terminated (see code replacement). However, due to the nature of these tables (just one-shot fully-qualified calls, no recursion or message-waiting construct), this is not expected to happen.

Finally, two extra table types exist:

- `const_bijective_table`, like a crossbreeding of `const_table` and `bijective_table`, to rely on module-supported const, bijective tables: a list of `{type_utils:permanent_term(), type_utils:permanent_term()}` entries can then provided so that a corresponding module (e.g. `foobar`) is generated (either in-memory or as a file) that allows to resolve any element of a pair into the other one, thanks to two functions, `foobar:get_first_for/1` and `foobar:get_second_for/1`, automatically defined in that module; this is especially useful for larger (const, bijective) tables; refer to `const_bijective_table_test.erl` for an example and a test thereof

- `const_bijective_topics` is the same as the previous type, except that it allows *multiple* of such (const, bijective) tables (named "topics" here) to be defined in the same module (e.g. `foobar`); for that, each of such tables is designated by a topic (an atom, like: `colour`, `bar_identifier` or `font_style`) that is associated to a declared list of associated entries (here alson each with no duplicate); then, for each of these topics (e.g. `colour`), two functions are automatically defined: `foobar:get_first_for_colour/1` and `foobar:get_second_for_colour/1`, returning respective elements of the specified pair, for the specified topic; refer to `const_bijective_topics_test.erl`

[10] Of course transient terms like PIDs, references, etc. cannot/should not stored in such tables.

for an example and a test thereof; the ability of defining multiple const, bijective tables in a single generated module can be useful typically when developping a binding (e.g. for a GUI) / translating protocols (e.g. between a third-party library and internal conventions); refer to Ceylan-Oceanic for an example thereof)

Other Datatypes They include `pair.erl`, `option_list.erl`, `preferences.erl`, `tree.erl`.

One may also refer for operations on:

- sets: `set_utils.erl`
- lists: `list_utils.erl`
- rings (i.e. circular buffers): `ring_utils.erl`
- binaries (i.e. raw binary information): `bin_utils.erl`

Pseudo-Builtin Types Such types, as `void/0` (for functions only useful for their side-effects - this happens!), `maybe/1` (`maybe(T)` is either `T` or `undefined`), `safe_maybe/1` (either `{just,T}` or `nothing`) and `fallible/{1,2}` (an operation either is successful and returns a result, or returns an error) are supported, thanks to the Myriad parse-transform.

Environments & Preferences

Principle An (application) **environment** is a server-like process that stores static or dynamic information (possibly initialised from an ETF file), as key/value entries (not unlike an ETS table), on behalf of an application or of a subset of its components, and makes it available to client processes.

Sharing of Data An environment stores a set of entries. An entry is designated by a key (an atom), associated to a value (that can be any term) in a pair.

Environments hold application-specific or component-specific data, obtained from any source (ETF file included); they may also start blank and be exclusively fed at runtime by the application or the components. Environments are used afterwards to maintain these pieces of data (read/write), before possibly storing them on file at application exit or component stop.

As a whole, an environment server can be seen as a process holding state information meant to be potentially common to various processes of a given application or component.

File Storage Environment data can optionally be read from or written to file(s) in the ETF format.

Example of content of an environment file:

```
{my_first_color, red}.
{myHeight, 1.80}.
{'My name', "Sylvester the cat"}.
```

Addressing Environment Servers The server process corresponding to an environment is locally registered; as a consequence it can be designated either directly through its PID or through its conventional (atom) registration name, like in:

```
environment:get(my_first_color, my_foobar_env_server).
```

No specific global registration of servers is made.

A (single) explicit start (with one of the `start*` functions) shall be preferred to implicit ones (typically triggered thanks to the `get*` functions) to avoid any risk of race conditions (should multiple processes attempt concurrently to create the same environment server), and also to be able to request that the server is also linked to the calling process.

An environment is best designated as a PID, otherwise as a registered name, otherwise from any filename that it uses.

About the Caching of Environment Entries For faster accesses (not involving any inter-process messaging), and if considering that their changes are rather infrequent (or never happening), at least some entries managed by an environment server may be cached directly in client processes.

In this case, the process dictionary of these clients is used to store the cached entries, and when updating a cached key from a client process the corresponding environment server is updated in turn. However any other client process caching that key will not be aware of this change until it requests an update to this environment server.

So a client process should cache a key mainly if no other is expected to update that key, i.e. typically if the associated value is const, or if this process is considered as the owner (sole controller) of that key (or if some other organisation ensures, possibly thanks to `sync/1`, that its cache is kept consistent with the corresponding environment server.

As soon as a key is declared to be cached, its value is set in the cache; there is thus always a value associated to a cached key (not a maybe-value), and thus cached values may be `undefined`.

Multiple environments may be used concurrently. A specific case of environment corresponds to the user preferences. See our `preferences` module for that, whose default settings file is `~/.ceylan-settings.etf`.

Resource Management

Principle Applications may have to manage all kinds of **data resources**, be them of classical resource types such as images or sounds, or be them specific to a project at hand.

The goal is to keep track of resources of all origins (e.g. read from file or network, or generated) in a *resource holder*.

These resources may be obtained:

- either from the filesystem, in which case their identifier is their (preferably binary) **path** that is relative to any holder-specific root directory (the recommended option) otherwise to the current directory, or absolute
- or from any other means, and then are designated thanks to a user-specified atom-based identifier

Resource Holders Myriad provides, through its `resource` module, two types of holders so that resources of interest can be obtained once, returned as often as needed, and stored for as long as wanted:

- resource **referentials**, which are process-local terms akin to associative tables
- resource **servers**, i.e. dedicated processes sharing resources (especially large-enough binaries) between any number of consumer processes

See also the `resource.hrl` include and the `resource_test` testing module.

File Formats

Basic File Formats A built-in very basic support for the CSV, for *Comma-Separated Values* (see `csv_utils`) and RDF (see `rdf_utils`) conventions is provided.

Most Usual, Standard File Formats Besides the support for XML, an optional support (as it depends on third-party prerequisites) is proposed for:

- JSON
- HDF5
- SQLite

Some useful information for **XML use**:

- Myriad's XML support is implemented by the `xml_utils` module (so one shall refer to `xml_utils.{e,h}rl` and `xml_utils_test.erl`), which relies on the built-in `xmerl` modules
- XML documents can be parsed from strings (see `string_to_xml/1`) or files (see `parse_xml_file/1`), and conversely can be serialised to strings (see `xml_to_string/{1,2}`)
- an XML document is made from a list of XML elements, that can exist as three different forms that can be freely mixed: as "simple-form", as IOLists and/or as XML (xmerl) records
- we recommend the use of the "simple-form", which should be sufficient for at least most cases

This last form is based on simple tags, used in order to easily have (Erlang) terms that are direct counterparts of XML tags.

For example the following two elements (respectively in simple-form and as an XML document) are equivalent (if using the default XML prolog):

```
XMLSimpleContent = [
  myFirstTag,
  {mySecondTag, [myNestedTag]},
  {myThirdTag, [{color, "red"}, {age, 71}], ["This is a text!"]}].
```

and:

```
<?xml version="1.0" encoding="utf-8" ?>
<myFirstTag/>
<mySecondTag><myNestedTag/></mySecondTag>
<myThirdTag color="red" age="71">This is a text!</myThirdTag>
```

Refer to the `xml_utils` module for further details.
Some useful information for **JSON use**:

- the nesting of elements shall be done thanks to (Erlang) maps, whose keys are binary strings (`text_utils:bin_string/0`); their order should not matter
- it may thus be convenient to add `-define(table_type, map_hashtable).` in a user module, so that the `table` pseudo-module can be relied upon when building a `json_term`, while being sure that the JSON parser at hand will be fed afterwards with the relevant datastructure
- no comments shall be specified (even though some parsers may be configured to support them)
- strings shall be specified as binary ones
- the actual JSON backend used are either jsx or jiffy; to better understand their (mostly common) mapping between Erlang and JSON, one may refer to the this section of the jsx documentation and to this one regarding jiffy

Example:

```
MyJSONTerm = table:add_entries([
  {<<"asset">>, #{<<"generator">> => <<"My Generator">>,
                  <<"version">> => <<"2.0">>}},
  {<<"other">>, 42}
                                 ], table:new()),

JSONString = json_utils:to_json(MyJSONTerm)
```

shall result in a JSON document like:

```
{
  "asset": {
    "generator": "My Generator",
    "version": "2.0"
  },
  "other": 42
}
```

Hint: the jq command-line tool may be very convenient in JSON contexts.
Refer to the Myriad-level Third-Party Dependencies section for further information.

For Pure Erlang uses: the ETF File Format For many needs in terms of Erlang internal data storage (ex: regarding configuration settings), we recommend the use of the file format that file:consult/1 can directly read, that we named, for reference purpose, `ETF` (for *Erlang Term Format*[11]). We recommend that ETF files have for extension `.etf`, like in: `~/.ceylan-settings.etf` (see also our support for user preferences).

ETF is just a text format for which:

- a line starting with a `%` character is considered to be a comment, and is thus ignored
- other lines are terminated by a dot, and correspond each to an Erlang term (ex: `{base_log_dir, "/var/log"}.`)

Note that no mute variable can be used there (e.g. `_Name="James Bond"` cannot be specified in such a file; only terms like `"James Bond"` can be parsed); so, in order to add any information of interest, one shall use comment lines instead.

Records are not known either; however they can be specified as tagged tuples (e.g. instead of specifying `#foo{ bar=7, ...}`, use `{foo, 7, ...}`).

See this example of a full ETF file.

A basic support for these ETF files is available in `file_utils:{read,write}_etf_file/*`.

If expecting to read UTF-8 content from such a file, it should:

- have been then opened for writing typically while including the `{encoding,utf8}` option, or have been written with content already properly encoded (maybe more reliable that way)
- start with a `%% -*- coding: utf-8 -*-` header

ETF files are notably used as **configuration files**. In this case following extra conventions apply:

- their extension is preferably changed from `.etf` to `.config`
- before each entry, a comment describing it in general terms shall be available, with typing information
- entries are pairs:
  - whose first element is an atom
  - their second element can be any value, typically of algebraic types; if a string value is included, for readability purpose it shall preferably be specified as a plain one (ex: `"James Bond"`) rather than a binary one (ex: `<<"James Bond">>`); it is up to the reading logic to accommodate both forms; it is tolerated to reference, in the comments of these configuration files, types that actually include *binary* strings (not plain ones, even though plain ones are used in the configuration files)

[11] Not to be mixed up with the Erlang External Term Format, which is used for serialisation.

To Export 3D Scenes A basic support of glTF (*Graphics Language Transmission Format*) version 2.0 has been implemented in `gltf_support.{hrl,erl}`.

The various elements associated to that model (scenes, nodes, meshes, primitives, materials, lights, cameras, buffers, buffer-views, accessors) can be handled from Erlang, in an already integrated way to Myriad's spatial services and conventions.

See the glTF 2.0 Reference Guide and the glTF 2.0 Specification for more information. See also our HOW-TO about 3D for both more general and practical considerations.

Regarding Data Exchange

Serialisation: Marshalling / Demarshalling

Purpose When trusted Erlang nodes and Erlang applications are to communicate, they are likely to rely on the (Erlang) External Term Format for that.

To communicate with other systems (non-Erlang and/or non-trusted) over a network stream (over a transport protocol such as TCP/IP), a common data-serialisation format must be chosen in order to marshall and demarshall the applicative data to be exchanged.

This format can be ad hoc (defined with one's conventions) or standard. We prefer here the latter solution, as a standard format favors interoperability and reduces tedious, error-prone transformations.

Moreover various well-supported standard options exist, like XDR, ASN.1, Protobuf (a.k.a. *Protocol Buffer*), Piqi and many others.

Choice of format The two formats that we thought were the most suitable and standard were **ASN.1** (with a proper, efficient encoding selected), or **Protobuff**.

As ASN.1 has been defined for long and is properly supported by Erlang (natively), and that there are apparently valid claims that Protobuf has some flaws, ASN.1 seemed to us the more relevant technical choice.

About ASN.1 Erlang supports, out of the box, three main ASN.1 encodings:

- BER (Basic Encoding Rules): a type-length-value encoding, too basic to be compact; its DER (for *Distinguished Encoding Rules*) variation is also available
- PER (*Packed Encoding Rules*): a bit-level serialisation stream, either aligned to byte boundaries (PER) or not (UPER, for *Unaligned PER*); if both are very compact and complex to marshall/demarshall, it is especially true for the size/processing trade-off of UPER
- JER (*JSON Encoding Rules*), hence based on JSON

Our preference goes towards first UPER, then PER. A strength of ASN.1 is the expected ability to switch encodings easily; so, should the OER encoding (*Octet Encoding Rules*; faster to decode/encode than BER and PER, and almost as compact as PER) be supported in the future, it could be adopted "transparently".

An issue of this approach is that, beyond Erlang, the (U)PER encoding does not seem so widely available as free software: besides commercial offers (like this one), some languages could be covered to some extent (ex: Python, Java with [1] or [2]), but for example no such solution could be found for the .NET language family (ex: for C#); also the complexity of the encoding may lead to solutions supporting only a subset of the standard.

So, at least for the moment, we chose Protobuf.

About Protobuf Compared to ASN.1 UPER, Protobuf is probably simpler/more limited, and less compact - yet also less demanding in terms of processing regarding (de)marshalling.

Albeit Protobuf is considerably more recent, implementations of it in free software are rather widely available in terms of languages, with reference implementations and third-party ones (example for .NET).

In the case of Erlang, Protobuf is not natively supported, yet various libraries offer such a support.

gpb seems to be the recommended option, this is therefore the backend that we retained. For increased performance, enif_protobuf could be considered as a possible drop-in replacement.

Our procedure to install `gpb`:

```
$ cd ~/Software/gpb
$ git clone git@github.com:tomas-abrahamsson/gpb.git
$ ln -s gpb gpb-current-install
$ cd gpb && make all
```

Then, so that `protoc-erl` is available on the shell, one may add in one's `~/.bashrc`:

```
# Erlang protobuf gpb support:
export GPB_ROOT="${HOME}/Software/gpb/gpb-current-install"
export PATH="${GPB_ROOT}/bin:${PATH}"
```

Our preferred settings (configurable, yet by default enforced natively by Myriad's build system) are: (between parentheses, the gbp API counterpart to the `protoc-erl` command-line options)

- `proto3` version rather than `proto2` (so `{proto_defs_version,3}`)
- messages shall be decoded as tuples/records rather than maps (so not specifying the `-maps` / `maps` option, not even `-mapfields-as-maps`) for a better compactness and a clearer, more statically-defined structure - even if it implies including the generated `*.hrl` files in the user code and complexifying the build (ex: tests having to compile with or without a Protobuff backend available, with or without generated headers; refer to `protobuf_support_test.erl` for a full, integrated example)
- decoded strings should be returned as binaries rather than plain ones (so specifying the `-strbin` / `strings_as_binaries` option)

- `-pkgs` / `use_packages` (and `{pkg_name, {prefix, "MyPackage"}}`) to prefix a message name by its package (regardless of the `.proto` filename in which it is defined)

- `-rename msg_fqname:snake_case` then `-rename msg_fqname:dots_to_underscores` (in that order), so that a message type named `Person` defined in package `myriad.protobuf.test` results in the definition of a `myriad_protobuf_test_person()` type and in a `#myriad_protobuf_test_person{}` record

- `-preserve-unknown-fields` (thus `preserve_unknown_fields`) will be set iff `EXECUTION_TARGET` has been set to `development` (`myriad_check_protobuf` is enabled), and in this case will be checked so that a warning trace is sent if decoding unknown fields

- `-MMD` / `list_deps_and_generate` to generate a `GNUmakedeps.protobuf` makefile tracing dependencies between message types

- `-v` / `verify` set to `never`, unless `EXECUTION_TARGET` has been set to `development` (`myriad_check_protobuf` is enabled), in which case it is set to `always`

- `-vdrp` / `verify_decode_required_present` set iff `EXECUTION_TARGET` has been set to `development` (`myriad_check_protobuf` is enabled)

- `-Werror` / `warnings_as_errors`, `-W1` / `return_warnings`, `return_errors` (preferably to their `report*` counterparts)

We prefer generating Protobuff (Erlang) accessors thanks to the command-line rather than driving the generating through a specific Erlang program relying on the gpb API.

See our `protobuf_support` module for further information.

This support may be enabled from Myriad's `GNUmakevars.inc`, thanks to the `USE_PROTOBUF` boolean variable that implies in turn the `USE_GPB` one.

One may also rely on our:

- `GNUmakerules-protobuf.inc`, in `src/data-management`, to include in turn any relevant dependency information; dependencies are by default automatically generated in a `GNUmakedeps.protobuf` file

- general explicit rules, for example `generate-protobuf` (to generate accessors), `info-protobuf` and `clean-protobuf` (to remove generated accessors)

- automatic rules, for example `make X.beam` when a `X.proto` exists in the current directory; applies our recommended settings)

One may note that:

- a Protobuff message, i.e. the (binary) serialised form of a term (here being a record), is generally smaller than this term (for example, `protobuf_support_test` reports a binary of 39 bytes, to be compared to the 112 bytes reported for the corresponding record/tuple)

- the encoding of the serialised form does not imply any specific obfuscation; for example binary strings comprised in the term to serialise may be directly readable from its binary serialisation, as clear text

References:

- general Protobuf Wikipedia presentation
- official page of Protobuf
- proto3 Language Guide
- gpb-related information:
  - command-line options: `protoc-erl -h`
  - gpb API documentation, notably the many options of gpb_compile documentation and the Erlang-Protobuff mapping

For Basic, Old-School Ciphering The spirit here is to go another route than modern public-key cryptography: the classic, basic, chained, symmetric cryptography techniques used in this section apply to contexts where a preliminary, safe exchange *can* happen between peers (ex: based on a real-life communication).

Then any number of passes of low-key, unfashioned algorithms (including one based on a Mealy machine) are applied to the data that is to cypher or decypher.

We believe that, should the corresponding shared "key" (the combination of parameterised transformations to apply on the data) remain uncompromised, the encrypted data is at least as safe as if cyphered with the current, modern algorithms (which may be, intentionally or not, flawed, or may be specifically threatened by potential progresses for example in terms of quantum computing).

So this is surely an instance of "security by obscurity", a pragmatic strategy (which may be used in conjunction with the "security by design" and "open security" ones) discouraged by standards bodies, yet in our opinion likely - for data of lesser importance - to resist well (as we do not expect then attackers to specifically target our very own set of measures, since the specific efforts incurred would not be outweighed by the expected gains).

We thus see such old-school ciphering as a complementary measure to the standard, ubiquitous measures whose effectiveness is difficult to assess for individuals and thus require some level of trust.

Refer to `cipher_utils` and its associated test for more details, and also to our mini-HOWTO regarding cybersecurity.

Support for Code Injection

It may be useful to decide, at compile-time, whether some code should be added / removed / transformed / generated based on **tokens** defined by the user.

This is done here thanks to the use of **conditional primitives** and associated **compilation defines** (sometimes designated as "macros", and typically specified in makefiles, with the `-D` flag).

These conditional primitives are gathered in the `cond_utils` module.

As an early example, so that a piece of code prints `Hello!` on the console when executed iff (*if and only if*) the `my_token` compilation token has been defined (through the `-Dmy_token` command-line flag), one may use:

```
cond_utils:if_defined(my_token, io:format("Hello!"))
```

Of course, as such a code injection is done at compilation-time, should compilation defines be modified the modules making use of the corresponding primitives shall be recompiled so that these changes are taken into account.

Let's enter a bit more in the details now.

Defining a token

A *token* (a compilation-time symbol) may or may not defined.

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

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

We recommend that layers built on top of Myriad define their own token for debug mode (ex: `foobar_debug_mode`), to be able to finely select appropriate debug modes (of course all kinds of modes and configuration settings can be considered as well).

Defining the code to inject

Based on the defined tokens, code may be injected; this code can be any Erlang expression, and the value to which it will evaluate (at runtime) can be used as any other value in the program.

Injecting a *single* expression (i.e. not multiple ones) is not a limitation: not only this single expression can be a function call (thus corresponding to arbitrarily many expressions), but more significantly a sequence of expressions (a.k.a. a *body*) can be nested in a `begin` / `end` block, making them a single expression[12].

[12] A previous implementation of `cond_utils` allowed to specify the code to inject either as an expression or as a *list* of expressions.

It was actually a mistake, as a single expression to return can be itself a list (ex: `["red", "blue"]`), which bears a different semantics and should not be interpreted as a list of expressions to evaluate. For example, the result from the code to inject may be bound to a variable, in which case we expect `A=["red", "blue"]` rather than `A="red", "blue"` (this latter term being constructed but not used).

Using tokens to enable code injection

Various primitives for *code injection* are available in the `cond_utils` (mostly pseudo-) module[13].

There is first `if_debug/1`, to be used as:

```
cond_utils:if_debug(EXPR_IF_IN_DEBUG_MODE)
```

Like in:

```
A = "Joe",
cond_utils:if_debug(io:format("Hello ~s!",[A]))
```

or, to illustrate expression blocks:

```
cond_utils:if_debug(begin
                        C=B+1,
                        io:format("Goodbye ~p",[C])
                    end)
```

These constructs will be replaced by the expression they specify for injection, at their location in the program, iff the `myriad_debug_mode` token has been defined, otherwise they will be replaced by nothing at all (hence with exactly *no* runtime penalty; and the result of the evaluation of `if_debug/1` is then not an expression).

Similarly, `if_defined/2`, used as:

```
cond_utils:if_defined(TOKEN, EXPR_IF_DEFINED)
```

will inject `EXPR_IF_DEFINED` if `TOKEN` has been defined (regardless of any value associated to this token), otherwise the `if_defined/2` call will be removed as a whole[14].

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

```
cond_utils:if_defined(TOKEN, EXPR_IF_DEFINED, EXPR_OTHERWISE)
```

For example:

```
% Older versions being less secure:
TLSSupportedVersions = cond_utils:if_defined(us_web_relaxed_security,
  ['tlsv1.3', 'tlsv1.2', 'tlsv1.1', 'tlsv1'],
  ['tlsv1.3'])
```

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

Note that a call to `if_defined/3` results thus in an expression.

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 set.

For `if_set_to/3`:

So the following code injection *is* faulty (a `begin/end` block was meant, not a list):

```
cond_utils:if_defined(my_token, [
    A = 1,
    io:format("Hello ~p!~n", [A])]),
```

(and moreover such code will trigger a compilation error, the `A` in `io:format/2` being reported as unbounded then)

[13] Their actual implementation lies in Myriad's parse transform.

[14] So `if_debug(EXPR)` behaves exactly as: `if_defined(myriad_debug_mode,EXPR)`.

```
cond_utils:if_defined(TOKEN, VALUE, EXPR_IF_SET_TO_THIS_VALUE)
```

will inject `EXPR_IF_SET_TO_THIS_VALUE` iff `TOKEN` has been defined and set to `VALUE`. As a result, the specified expression will not be injected if `some_token` has been set to another value, or not been defined at all.

Usage example, `-Dsome_token=42` having possibly been defined beforehand:

```
cond_utils:if_set_to(some_token,42, SomePid ! hello)])
```

As for `if_set_to/4`, in:

```
cond_utils:if_set_to(TOKEN, VALUE, EXPR_IF_SET_TO_THIS_VALUE, EXPR_OTHERWISE)
```

`EXPR_IF_SET_TO_THIS_VALUE` will be injected iff `TOKEN` has been defined and set to `VALUE`, otherwise (not set or set to a different value) `EXPR_OTHERWISE` will be.

Example:

```
Level = cond_utils:if_set_to(my_token, foobar_enabled, 1.0, 0.0) + 4.5
```

A similar construct in spirit is `switch_execution_target/2`, which will, depending on the current build-time execution target, inject a corresponding expression:

```
cond_utils:switch_execution_target(EXPR_IF_IN_DEVELOPMENT_MODE, EXPR_IF_IN_PRODUCTION_
```

So if the current execution target is development, the compilation will inject `EXPR_IF_IN_DEVELOPMENT_MODE`, otherwise `EXPR_IF_IN_PRODUCTION_MODE` will be.

Example:

```
io:format( "We are in ~ts mode.",
    [cond_utils:switch_execution_target("development", "production")])
```

Finally, the `switch_set_to/{2,3}` primitives allow to generalise these `if`-like constructs, with one among any number of code branches selected based on the build-time value of a token, possibly with defaults (should the token not be defined at all, or defined to a value that is not among the ones associated to a code branch).

For that we specify a list of pairs, each made of a value and of the corresponding expression to be injected if the actual token matches that value, like in:

```
cond_utils:switch_set_to(TOKEN, [
       {VALUE_1, EXPR_1},
       {VALUE_2, EXPR_2},
       % [...]
       {VALUE_N, EXPR_N}])
```

For example:

```
cond_utils:switch_set_to(my_token, [
     {my_first_value, io:format("Hello!")},
     {my_second_value, begin f(), g(X,debug), h() end},
     {some_third_value, a(X,Y)}])
```

A compilation-time error will be raised if `my_token` is not set, or if it is set to none of the declared values (i.e. not in `[my_first_value, my_second_value, some_third_value]`).

A variation of this primitive exists that applies a default token value if none was, or if the token was set to a value that is not listed among any of the ones designating a code branch, like in:

```
cond_utils:switch_set_to(TOKEN,
                           [ {VALUE_1, EXPR_1},
                             {VALUE_2, EXPR_2},
                             % [...]
                             {VALUE_N, EXPR_N}],
                           DEFAULT_VALUE)
```

As always with primitives that define a default, alternate branch, they always inject an expression and thus can be considered as such.

For example:

```
ModuleFilename = atom_to_list( cond_utils:switch_set_to(some_token,
               [{1, foo}, {14, bar}, {20, hello}], 14) ++ ".erl"
```

Here, if `some_token` is not defined, or defined to a value that is neither `1`, `14` or `20`, then the `14` default value applies, and thus `ModuleFilename` is set to `"bar.erl"`.

Refer to cond_utils_test.erl for further usage examples.

Controlling assertions

It may be convenient that, depending on a compile-time token (ex: in debug mode, typically triggered thanks to the `-Dmyriad_debug_mode` compilation flag), *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 only `myriad_debug_mode`) 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()`).

Finally, if assertions are too limited (ex: because they lead to unused variables depending on a token being defined or not), using one of the `cond_utils:if*` primitives relying on two branches (one expression if a condition is true, another if not) should sufficient to overcome such issue.

Usage Hints

For tokens, at least currently they must be defined as immediate values (atoms); even using a mute variable, like for the `_Default=my_token` expression, or a variable, is not supported (at least yet).

Note that, for primitives that may not inject code at all (ex: `if_debug/1`), if their conditions are not fulfilled, the specified conditional code is dismissed as a whole, it is not even replaced for example by an `ok` atom; this may matter if this conditional is the only expression in a case clause for example, in which case a compilation failure like "*internal error in core; crash reason: function_ clause in function v3_ core:cexprs/3 called as v3_ core:cexprs[...]*" will be reported (the compiler sees unexpectedly a clause not having even a single expression).

A related issue may happen when switching conditional flags: it will select/deselect in-code expressions at compile time, and may lead functions and/or variables to become unused, and thus may trigger at least warnings[15].

For **functions** that could become unused due to the conditional setting of a token, the compiler could certainly be silenced by exporting them; yet a better approach is surely to use:

```
-compile({nowarn_unused_function,my_func/3}).
```

or:

```
-compile({nowarn_unused_function,[my_func/3, my_other_func/0]}).
```

As for **variables**, should A, B or C be reported as unused if `some_token` was not set, then the `basic_utils:ignore_unused/1` function (mostly a no-op) could be of use:

```
[...]
cond_utils:if_defined(some_token,
                      f(A, B, C),
                      basic_utils:ignore_unused([A, B, C])),
[...]
```

Alternatively, `nowarn_unused_vars` could be used instead, at least in some modules.

[15] Warnings that we prefer promoting to errors, as they constitute a *very* convenient safety net.

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.

MyriadGUI: 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.

The spirit of **MyriadGUI** is to offer, as much as possible, a high-level API (refer to the `ui` module) that can be seamlessly instrumented at runtime by different backends, depending on availability (ex: is this dependency installed?) and context (ex: is the program running in a terminal, or are graphical outputs possible?).

Unless the user forces the use of a given backend, the most advanced one that is locally available will be automatically selected.

An objective is to shelter user code from:

- the actual UI backend that will be selected ultimately on a given host
- the rise and fall of the various backends (thinking for example to `gs` having been quickly deprecated in favour of `wx`); the idea is then that any new backend could be integrated, with *little to no change* in already-written code relying on MyriadGUI

Of course not all features of all backends can be integrated (they have not the same expressivity, a common base must be enforced[16]) and creating a uniform interface over all sorts of vastly different ways of displaying and interacting with the user would require a lot of efforts. So MyriadGUI follows a pragmatic route: defining first basic, relevant, user-centric conventions and services able to cover most needs and to federate (hopefully) most backends, and to incrementally augment its implementation coverage on a per-need basis. As a consequence, efforts have been made so that adding any lacking element can be done at low cost.

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, with colours and text-based widgets)
- or **graphical**, with `gui`
- (and/or, in a possible future, **audio**, with a `audio_gui` that could be added)

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:

[16] Yet optional, additional features may be defined, and each backend may choose to provide them or ignore them.

- 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`; similarly `gui` is richer than `term_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
$ 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.

It is 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
```

It is 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`

The `gui` modules provide features like 2D/3D rendering, event handling, input management (keyboard/mouse), canvas services (basic or OpenGL), and the various related staples (management of images, texts and fonts, colors, window manager, etc.); refer to the gui sources for more complete information.

For Classical 2D Applications

Base GUI Backend This interface used to rely on (now deprecated) `gs`, and now relies on wx[17][18] (a port of wxWidgets, which belongs to the same category as GTK or Qt). For the base dialogs, Zenity could have been an option.

We also borrowed elements from the truly impressive Wings3D (see also our HOWTO section about it) modeller, and also on the remarkable libSDL (2.0) library together with its esdl2 Erlang binding.

If having very demanding 2D needs, one may refer to the 3D services section (as it is meant to be hardware-accelerated, and the 2D services are a special cases thereof).

> **Note**
>
> Currently MyriadGUI does not adhere yet to the `ui` conventions, but it will ultimately. MyriadGUI already provides many lower-level services and offers a graphical API (currently on top of `wx`; see our HOWTO for some information regarding that backend) that can be used in order to develop one's GUI application hopefully in a future-proof way.

As a consequence, wxWidgets must be available on the host (otherwise a `{load_driver,"No driver found"}` exception will be raised on GUI start). This should correspond to the `wxgtk3` Arch Linux package, or the `libwxgtk3.0-dev` Debian one. This can be tested by executing `wx-config --version` on a shell.

`wxWidgets` must be installed *prior* to building Erlang, so that it is detected by its configuration script and a proper `wx` module can be used afterwards. Running then `wx:demo()` is a good test of the actual support.

Purpose of `gui` 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.; so none of their includes, records,

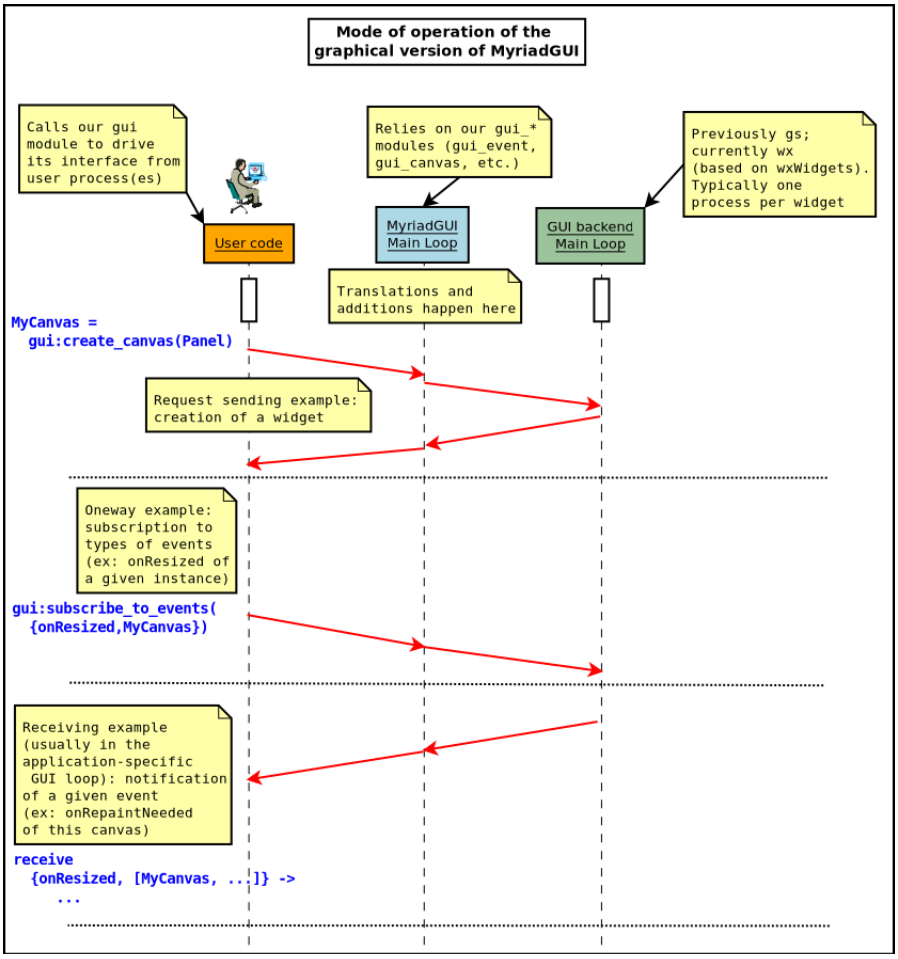
[17] What are the main differences between MyriadGUI and wx? The MyriadGUI API is backend-agnostic (no trace of wx when using it), a bit higher-level (ex: user-defined widget identifiers being atoms rather than integer constants; relying on more flexible options; integrating a few workarounds), and based on fewer modules. However, as a strict subset of wx, it is by design less complete - yet it is quite easy to extend on a per-need basis.

[18] Maybe later it will be based on HTML 5 (although we are not big fans of light clients and of using browsers for everything), possibly relying some day for that on the Nitrogen web framework, on N2O or on any other relevant HTML5 framework.

types or functions leak in the user realm), to avoid that user programs become obsolete too quickly because of the UI backend they rely on.

So for example the messages received by the user programs do not mention `wx`, and respect only MyriadGUI conventions. These conventions are in line with the WOOPER ones, enabling (in a fully optional manner) the user code to rely on WOOPER if wanted.

The usual mode of operation is the following:



1. From a user process (a test, an application, etc.), the GUI support is first started, with `gui:start/{0,1}`
2. Then the various widgets (windows, frames, panels, buttons, etc.) are created (ex: thanks to `MainFrame = gui:create_frame(...)` and the user process subscribes to the events it is interested in (as a combination of an event type and a widget-as-an-event-emitter; for example:

   `gui:subscribe_to_events({onWindowClosed, MainFrame})`
3. The user process also triggers any relevant operation (ex: clearing widgets, setting various parameters), generally shows at least a main frame and

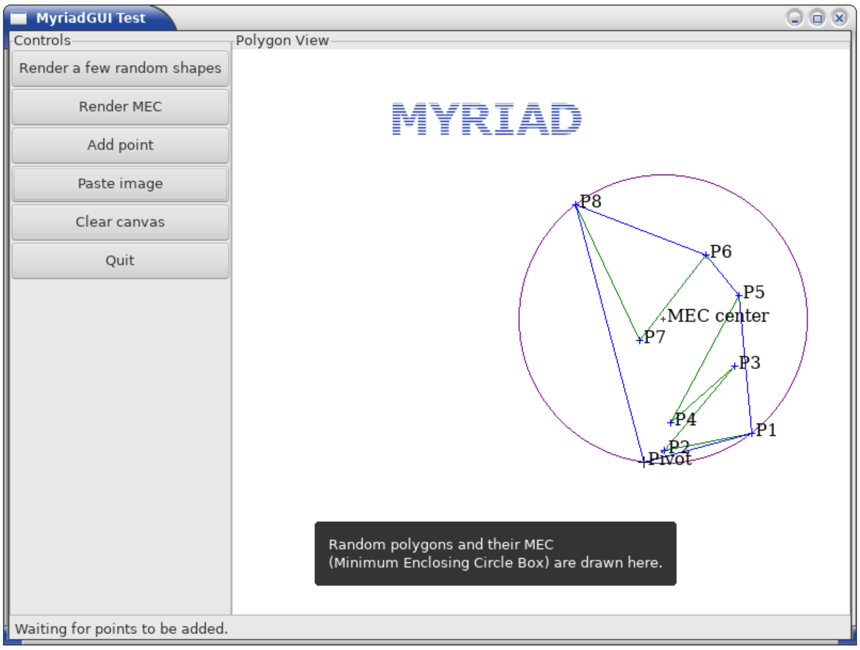
records the GUI state that it needs for future use (typically containing at least the MyriadGUI references of the widgets that it created)

4. Then the user process enters its own (GUI-specific) main loop, from which it will receive the events that it subscribed to, and to which it will react by performing application-specific operations and/or GUI-related operations (creating, modifying, deleting widgets). Generally at least one condition is defined in order to leave that main loop and stop the GUI (`gui:stop/0`)

Such a scheme based on a "man-in-the-middle" (the MyriadGUI process) is necessary to abstract out for example the types of messages induced by a given GUI backend. If performances should not be an issue for user interaction, the integration must be carefully designed, notably because a 3-actor cooperation (user code, MyriadGUI one, backend one) opens the possibility of race conditions to occur (notably some operations, like event subscribing, must then be made synchronous, as the user process may trigger direct interactions with the backend; see implementation notes for more details).

Refer to the gui_overall_test.erl and lorenz_test.erl test full, executable usage examples thereof.

Here is a screenshot of the former test, where a random polygon (in green) is generated, for which are determined both the convex hull (in blue) and the MEC (*Minimum Enclosing Circle*, in purple):

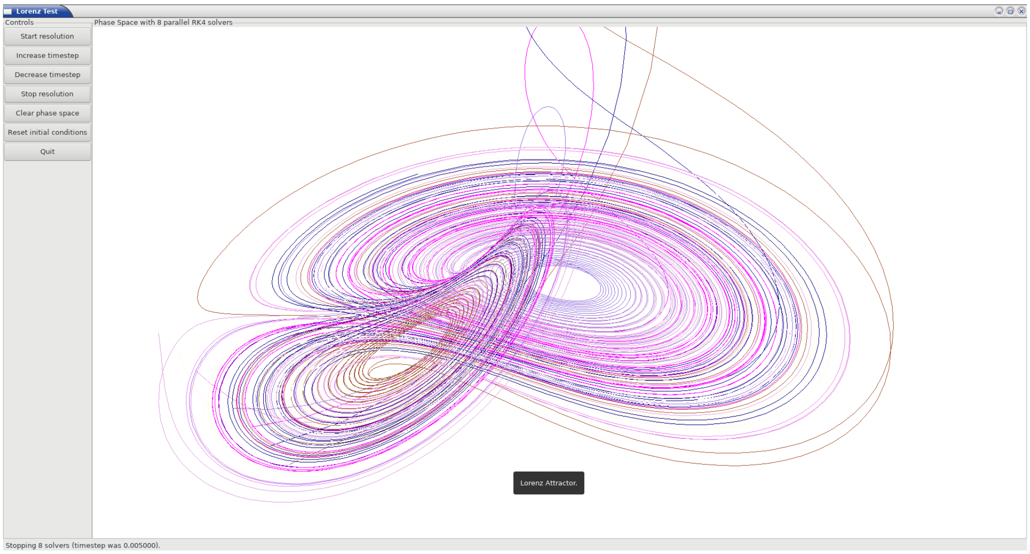


Defining `gui` as an interface between the user code and a backend also allows to enrich said backend[19].

[19] For example, we needed to operate on a plain canvas, whereas `wx` (as we understand it) offers only panels with bitmaps (with `wxDC`, `wxWindowDC`, `wxMemoryDC`, etc.), with no possibility to subclass them in order to add them features. So MyriadGUI transparently introduced `gui_canvas` to offer extended canvas services.

These services are 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`) and will be enriched over time, on a per-need basis.

This last `lorenz_test.erl` offers another complete example:



For 3D Applications

Purpose In order to render 3D content, Myriad relies on OpenGL, a standard, cross-platform, uniform and well-designed programming interface that enables the use of video cards in order to deport most of the (2D or 3D) heavy-lifting there.

Sophisticated 3D rendering is not necessarily an area where Erlang shines (perhaps, on the context of a client/server multimedia application, the client could rely on an engine like Godot instead), yet at least some level of rendering capabilities is convenient whenever performing 3D computations, implementing a server-side 3D logic, processing meshes, etc.

One may refer to our 3D mini-HOWTO for general information regarding these topics.

Prerequisites So a prerequisite is that the local host enjoys at least some kind of **OpenGL support**, either in software or, most preferably, with an hardware acceleration.

Just run our `gui_opengl_integration_test.erl` test to have the detected local configuration examined. One should refer to our HOWTO section about 3D operating system support for detailed information and troubleshooting guidelines.

As for the **Erlang side** of this OpenGL support, one may refer to this section to ensure that the Erlang build at hand has indeed its OpenGL support enabled.

3D Services User API

The Myriad OpenGL utilities are defined in the `gui_opengl` module.

Shaders can be defined, in GLSL (see this page for more information).

Myriad recommends using the `vertex.glsl` extension for vertex shaders, the `.tess-ctrl.glsl` one for tessellation control shaders and `.tess-eval.glsl` for tessellation evaluation ones, `.geometry.glsl` for geometry shaders, `fragment.glsl` for fragment shaders and finally `.compute.glsl` for compute shaders.

The many OpenGL defines are available when having included `gui_opengl.hrl` (ex: as `?GL_QUAD_STRIP`).

These utilities directly relate to Myriad's spatial services and conventions and to its support of the glTF file format.

To manage larger 3D scenes, a basic support of octrees is also available (see octree.erl); following conventions apply:

Various tests offer usage examples of the MyriadGUI API for 3D rendering:

- `gui_opengl_minimal_test.erl` runs a minimal test showcasing the proper local OpenGL support, based on normalised coordinates (in `[0.0,1.0]`)
- `gui_opengl_2D_test.erl` is a 2D test operating with absolute (non-normalised) coordinates
- `gui_opengl_integration_test.erl` demonstrates more features (quadrics, textures, etc.)
- `gui_opengl_mvc_test.erl` proposes a MVC architecture (Model-View-Controller) where these three elements are uncoupled in separate processes yet are properly interlinked, the view relying on the MyriadGUI OpenGL support
- `gui_opengl_minimal_shader_test.erl` showcases the use of more recent OpenGL APIs (3.3 core[20]), with GLSL shaders defined in `gui_opengl_minimal_shader.{vertex,fragm`

Note

Almost all OpenGL operations require that an OpenGL context already exists. When it is done, all GL/GLU operations can be done as usual. So the point of MyriadGUI here is mostly to create a suitable OpenGL context, to offer a few additional, higher-level, stricter constructs to ease the integration and use (ex: for the compilation of the various types of shaders and the linking of GLSL programs), and to connect this rendering capability to the rest of the GUI (ex: regarding event management).

Modern OpenGL is supported (ex: version 4.6), even though the compatibility context allows to use the API of OpenGL version 1.1.

See the HOWTO section about OpenGL for more explanations.

[20]Not compatible with all GPUs; notably Intel ones may only support older versions (ex: 2.1).

Configuration

In terms of error management, extensive verifications will apply iff the `myriad_check_opengl_support` flag is set.

Setting the `myriad_debug_opengl_support` flag will result in more runtime information to be reported.

Internal Implementation

MyriadGUI is a wrapper on top of wx. What are the main differences between MyriadGUI and wx?

- preferred namings introduced (ex: `onWindowClosed` events found clearer than `close_window` ones)
- widget identifiers are user-defined atoms in MyriadGUI (ex: `my_widget_id`) rather than numerical constants (ex: `-define(MY_WIDGET_ID, 2051)`) that have, with wx, to be defined, shared, uniquified accross user modules
- by default, events will propagate or be trapped by user-defined handlers depending on the type of these events (most of them being propagated by default; of course the user is able to override these defaults, either at subscription-time - using the `propagate_event` or `trap_event` option, or in one's handler - using the `gui:propagate_event/1` or `gui:trap_event/1` function); this contrasts with wx, in which by default all subscribed events are trapped, regardless of their type (then forgetting to propagate them explicitly may result in built-in mechanisms of wx to be disabled, like when resizing)
- code using MyriadGUI will not depend on wx, opening the possibility that, should the main Erlang GUI backend change, user code is nevertheless preserved

See also our little Using wx HOWTO.

Regarding hardware acceleration, the MyriadGUI 2D/3D services rely on the related Erlang-native modules, namely gl and glu, which are NIF-based bindings to the local OpenGL library.

As for the `wx` module (see the wx availability section), it provides a convenient solution in order to create a suitable OpenGL context.

esdl used to be another solution to obtain an OpenGL context; it may be revived some day, as SDL - i.e. *Simple DirectMedia Layer* - is still striving, and offers a full (yet low-level) access to multimedia and input devices; not all applications may have use of the rest of `wx`.

These Erlang-native services can be easily tested by running `wx:demo()` from any Erlang shell and selecting then `gl` in the left example menu.

These platform-specific / backend-specific (ex: wx or not, and which version thereof, ex: wxWidget 2.8 vs 3.0 API) services shall remain fully invisible from MyriadGUI user code, so that it remains sheltered for good from any change at their level.

The goal is to wrap only the dependencies that may change in the future (ex: wx); doing so for the ones considered (for good reasons) stable (such as gl or glu) would have no specific interest.

For Multimedia Applications Currently we provide only very preliminary support thereof with `audio_utils`; for sound and music playback, refer to the audio section for more details.

Speech synthesis (TTS, *Text-to-Speech*) is available thanks to `speech_utils.erl`. In practice, for best results, the actual speech generation is delegated to cloud-based providers making use of AI (neural voices) for best fluidity.

The input text shall preferably comply with SSML, typically so that it can be enriched with phonemes and prosody hints.

One can listen to this French speech and this English one (most browsers are now able to playback Opus content), that have been both generated by `speech_utils_test.erl`.

Facilities to manage logical speeches (i.e. speeches designated by a base name such as `hello` and declined in as many locales as needed) are available (see `speech_support:logical_speech/0`), as well as for related containers (see `speech_support:speech_referential/0`).

For Interactive Applications Beyond the rendering of multimedia content, user interfaces have to **collect inputs from the user**, typically through mice, keyboards and joysticks.

Formerly, a port of SDL, esdl, was the best option, now using `wx` for that is recommended, as, through this port, the various input devices can be managed (at least to a large extent).

Audio User Interface

If the 2D/3D rendering can be done through `wx`, apparently the **audio capabilities** (ex: [1], [2]) of wxWidgets have not been made available to Erlang.

So an Erlang program needing audio output (ex: sound special effects, musics) and/or input (ex: microphone) will have to rely on another option, possibly in link, for audio rendering, to 3D-ready eopenal - an (Erlang) binding of OpenAL, or to a lower-level SDL-based solution. Contributions welcome!

Currently only very basic support for audio output is available, as `audio_utils:playback_file/{2,3}`.

See also our support for speech synthesis.

Myriad's Helper Shell Scripts

A small set of shell scripts has been defined, in `myriad/src/scripts`, to provide generic facilities useful in the context of Myriad[21].

Erlang-Dedicated Scripts

Searching for Erlang elements

ergrep To search for a text pattern through an Erlang source tree.

> Usage: ergrep [-v|--verbose] [-q|--quiet] [-f|--filenames-only] [-i|--insensitive] <Expression to be found in sources> [TARGET_BASE_DIR]: recursive grep in Erlang source files, either from the TARGET_BASE_DIR directory, if specified, otherwise from the current directory.
>
> Options:
>
> -v or --verbose: be specifically verbose
>
> -q or --quiet: be specifically quiet, just listing matches
>
> -f or --filenames-only: display only filenames, not also the matched patterns, and if there are multiple matches in the same file, its filename will be output only once (implies quiet); useful for scripts
>
> -i or --insensitive: perform case-insensitive searches in the content of files, and also in the searched Erlang filenames
>
> Example: ergrep -i 'list_to_form(' /tmp

find-type-definition.sh To search for the definition of a type of interest.

> Usage: find-type-definition.sh [-h|--help] A_TYPE [A_DIR]: attempts to find the definition of the specified Erlang type from the specified directory (if any), otherwise from the current one.

find-function-specification.sh To search for a function spec of interest.

> Usage: find-function-specification.sh [-h|--help] A_FUNCTION_NAME [A_DIR]: attempts to find the type specification for the specified Erlang function (even if the name is partial and/or includes an arity) from the specified directory (if any), otherwise from the current one.

find-record-definition.sh To search for the definition of a record of interest.

[21] For a more general collection of (different) scripts, one may refer to Ceylan-Hull, notably the ones to facilitate development.

Usage: find-record-definition.sh [-h|--help] A_RECORD_NAME [A_DIR]: attempts to find the definition of the specified Erlang record from the specified directory (if any), otherwise from the current one.

Regarding Typing

list-available-types.sh Lists all Erlang types found in specified tree.

Usage: list-available-types.sh [-h|--help] [ROOT_DIR]: lists all types (according to Erlang type specifications) defined from the ROOT_DIR directory (if specified) or from the current directory.

add-deduced-type-specs.escript Generates and adds the type specification of all functions in specified module(s).

Usage: add-deduced-type-specs.escript FS_ELEMENT

Adds, for each selected BEAM file (either specified directly as a file, or found recursively from a specified directory), in the corresponding source file(s), for each function, the type specification that could be deduced from its current implementation.

FS_ELEMENT is either the path of a BEAM file or a directory that will be scanned recursively for BEAM files.
Note that BEAM files must be already compiled, and with debug information (see the '+debug_info' compile flag).
Note also that generating type specs this way may not a good practice.

Regarding Basic Performance Measurement

etop.sh Monitors on the console the busiest Erlang processes of a VM.

Usage: etop.sh [-node NODE_NAME] [-setcookie COOKIE]: shows on the console the activity of the Erlang processes on specified Erlang node (enter CTRL-C twice to exit).
Example: etop.sh -node foobar@baz.org -setcookie 'my cookie'

benchmark-command.escript Returns a mean resource consumption for the specified shell command (one may prefer relying on benchmark-command.sh).

Usage: benchmark-command.escript COMMAND: returns a mean resource consumption for the specified shell command.

Example: benchmark-command.escript "my_script.sh 1 2"

Miscellaneous

make-code-stats.sh Outputs basic statistics about an Erlang code base.

Usage: make-code-stats.sh [-h|--help] [SOURCE_DIRECTORY]: evaluates various simple metrics of the Erlang code found from any specified root directory, otherwise from the current one.

launch-erl.sh The only shell script on which we rely in order to launch an Erlang VM.

Usage: launch-erl.sh [-v] [-c a_cookie] [--sn a_short_node_name | --ln a_long_node_name | --nn an_ignored_node_name] [--tcp-range min_port max_port] [--epmd-port new_port] [--fqdn a_fqdn] [--max-process-count max_count] [--busy-limit kb_size] [--async-thread-count thread_count] [--background] [--non-interactive] [--eval an_expression] [--no-auto-start] [-h|--help] [--beam-dir a_path] [--beam-paths path_1 path_2] [-start-verbatim-options [...]]: launches the Erlang interpreter with specified settings.

Detailed options:

- -v: be verbose
- -c a_cookie: specify a cookie, otherwise no cookie will be specifically set
- --sn a_short_node_name: distributed node using specified short name (ex: 'my_short_name')
- --ln a_long_node_name: distributed node using specified long name (ex: 'my_long_name')
- --nn an_ignored_node_name: non-distributed node, specified name ignored (useful to just switch the naming options)
- --tcp-range min_port max_port: specify a TCP port range for inter-node communication (useful for firewalling issues)
- --epmd-port new_port: specify a specific EPMD port (default: 4369); only relevant if the VM is to be distributed (using short or long names), initially or at runtime
- --fqdn a_fqdn: specify the FQDN to be used
- --max-process-count max_count: specify the maximum number of processes per VM (default: 400000)
- --busy-limit size: specify the distribution buffer busy limit, in kB (default: 1024)
- --async-thread-count thread_count: specify the number of asynchronous threads for driver calls (default: 128)
- --background: run the launched interpreter in the background (ideal to run as a daemon, ex: on a server)
- --daemon: run the node as a daemon (relies on run_erl and implies --background)

--non-interactive: run the launched interpreter with no shell nor input reading (ideal to run through a job manager, ex: on a cluster)

--eval 'an Erlang expression': start by evaluating this expression

--no-auto-start: disable the automatic execution at VM start-up

-h or --help: display this help

--beam-dir a_path: adds specified directory to the path searched for beam files (multiple --beam-dir options can be specified)

--beam-paths first_path second_path ...: adds specified directories to the path searched for beam files (multiple paths can be specified; must be the last option)

--log-dir: specify the directory in which the VM logs (if using run_erl) shall be written

Other options will be passed 'as are' to the interpreter with a warning, except if they are listed after a '-start-verbatim-options' option, in which case they will passed with no warning.

If neither '--sn' nor '--ln' is specified, then the node will not be a distributed one.

Example: launch-erl.sh -v --ln ceylan --eval 'foobar_test:run()'

show-xml-file.escript Displays the content of the specified XML file.

Usage: show_xml_file.escript XML_FILE_PATH
Displays sequentially in a {name,Value} tree the structure of specified XML file (XML elements along with their XML attributes).

More General Scripts

To generate documentation These scripts are mostly unrelated to Erlang, yet are useful to be available from our most basic layer (Myriad).

generate-docutils.sh Generates a proper PDF and/or HTML file from specified RST (reStructuredText) one (main, standalone script).

Usage: generate-docutils.sh <target rst file>
[--pdf|--all|<comma-separated path(s) to CSS file to be used, ex: common/css/XXX.css,other.css>] [--icon-file ICON_FILENAME]

Generates a final document from specified docutils source file (*.rst).

If '--pdf' is specified, a PDF will be created, if '--all' is specified, all output formats (i.e. HTML and PDF) will be created, otherwise HTML files only will be generated, using any specified CSS file.

generate-pdf-from-rst.sh Generates a proper PDF and/or HTML file from specified RST (reStructuredText) one; the previous `generate-docutils.sh` script is often preferred to this one, which depends on Myriad.

Usage: generate-pdf-from-rst.sh RST_FILE: generates a PDF file from the specified RST file, overwriting any past file with that name.

Ex: 'generate-pdf-from-rst.sh my_file.rst' will attempt to generate a new 'my_file.pdf' file.

Script-based Apps

These shell scripts are actually user-facing shell interfaces that plug directly on some more involved Erlang programs, i.e. applications that are available here.

generate-password.sh Generates a proper random password respecting various rules, whose simple application can be transparently checked (probably at least more easily audited than most password managers - thus maybe more trustable).

Usage: generate-password.escript [-a ALPHABET|--alphabet ALPHABET]
[-l MIN_LEN MAX_LEN|--length MIN_LEN MAX_LEN]
[-h|--help]

Generates a suitable password, where:
- ALPHABET designates the set of characters to draw from (default one being 'extended'), among:
* 'base': alphanumeric letters, all cases [A-Za-z0-9]
* 'extended': 'base' + basic punctuation (i.e. '[](){}:,;-_.!?')
* 'full': 'base' + all punctuation (i.e. basic + '"'@ /&$*^%=+|')
- MIN_LEN and MAX_LEN are the respective minimum and maximum numbers of characters
(bounds included) used to generate this password [default: between 15 and 20]

See also: the security section of Ceylan-Hull, for more general guidelines and tooling regarding the proper management of credentials.

merge.sh Helps merging efficiently and reliably file trees; it is actually a rather involved text-based application that allows scanning/comparing/merging trees, typically in order to deduplicate file hierarchies that were exact copies once, yet may have since then diverged.

Usage: following operations can be triggered:

- 'merge.sh --input INPUT_TREE --reference REFERENCE_TREE'
- 'merge.sh --scan A_TREE'
- 'merge.sh --rescan A_TREE'
- 'merge.sh --resync A_TREE'
- 'merge.sh --uniquify A_TREE'
- 'merge.sh -h' or 'merge.sh --help'

Ensures, for the first form, that all the changes in a possibly more up-to-date, "newer" tree (INPUT_TREE) are merged back to the reference tree (REFERENCE_TREE), from which the first tree may have derived. Once executed, only a refreshed, complemented reference tree will exist, as the input tree will have been removed: all its original content (i.e. its content that was not already in the reference tree) will have been transferred in the reference tree.

In the reference tree, in-tree duplicated content will be either kept as it is, or removed as a whole (to keep only one copy thereof), or replaced by symbolic links in order to keep only a single reference version of each actual content.

At the root of the reference tree, a '.merge-tree.cache' file will be stored, in order to avoid any later recomputations of the checksums of the files that it contains, should they have not changed. As a result, once a merge is done, the reference tree may contain an uniquified version of the union of the two specified trees, and the input tree will not exist anymore after the merge.

For the second form (--scan option), the specified tree will simply be inspected for duplicates, and a corresponding '.merge-tree.cache' file will be created at its root (to be potentially reused by a later operation).

For the third form (--rescan option), an attempt to rebuild an updated '.merge-tree.cache' file will be performed, computing only the checksum of the files that were not already referenced, or whose timestamp or size changed.

For the fourth form (--resync option), a rebuild even lighter than the previous rescan of '.merge-tree.cache' will be done, checking only sizes (not timestamps), and updating these timestamps.

For the fifth form (--uniquify option), the specified tree will be scanned first (see the corresponding operation), and

For the fifth form (--uniquify option), the specified tree will be scanned first (see the corresponding operation), and
then the user will be offered various actions regarding found duplicates (being kept as are, or removed, or replaced with
symbolic links), and once done a corresponding up-to-date '.merge-tree.cache' file will be created at its root (to be
potentially reused by a later operation).

For the sixth form (-h or --help option), displays this help.

Note that the --base-dir A_BASE_DIR option can be specified by the user to designate the base directory of all relative
paths mentioned. When a cache file is found, it can be either ignored (and thus recreated) or re-used, either as it is or
after a weak check, where only file existence, sizes and timestamps are then verified (not checksums).

See also: the `test-all` target of the merge-related makefile, to give it a try before applying such procedure to your data of interest.

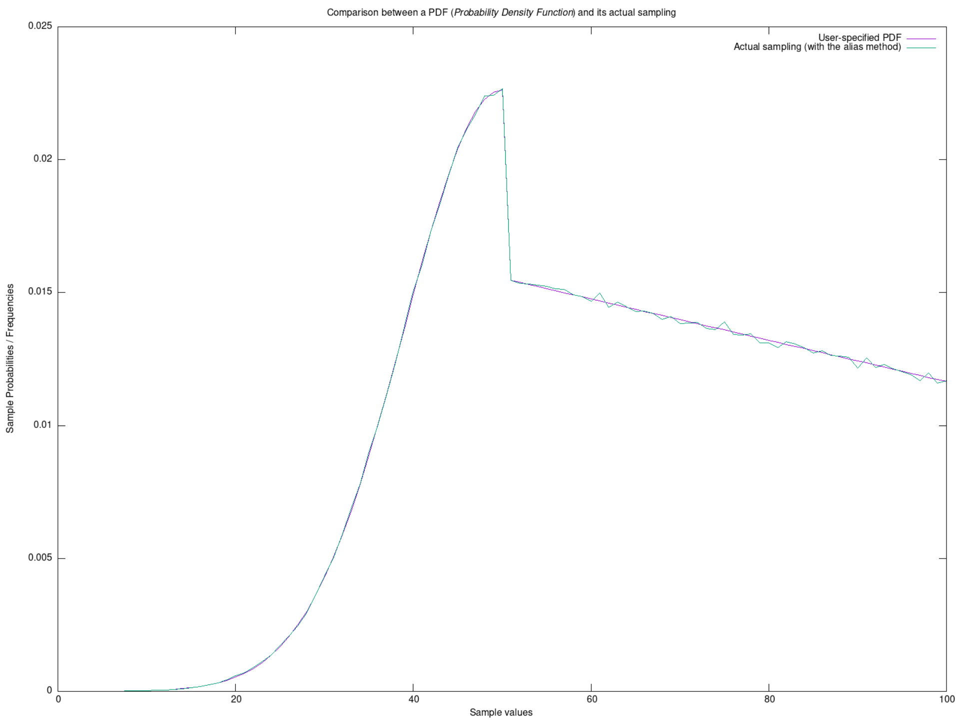
Utility Toolbox

> **Note**
>
> This part is the **actual core** of the `Ceylan-Myriad` library: a toolbox comprising many helper functions (with their tests), defined in the `myriad/src/{utils,data-management}` directories, often providing richer, 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` (for a more collection of helpers) or `_support` (for a service deserving more integration), and include:

- many **basic, general-purpose services**, defined in `basic_utils.erl`, regarding:
  - the base types we defined
  - notifications
  - message handling
  - many miscellaneous functions
- 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
- a few **test-related facilities**, in `test_facilities.erl`
- 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`); note that we now recommend not to open files with a specific encoding being set, but instead to encode the content before any writing thereof; refer to the `Regarding encodings and Unicode` section in `file_utils.erl` for further information
- services to manage identifiers of various sorts, including sortable ones (`id_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}`)
- support for **network**-related concerns (`net_utils.erl.{e,h}rl`)

- services to offer **randomness** (`random_utils.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.; uniform sampling can be performed like non-uniform one, then based on a user-specified probability distribution, as shown by the following example (generated by `random_utils_test.erl` after 1 million draws taken from a test, arbitrary probability density function - here a Gaussian on its first half domain, and an affin function on its second half):



- very little support of **RDF** operations, standing for Resource Description Framework (`rdf_utils.erl`)
- support for **XML** (`xml_utils.{e,h}rl`, based on built-in `xmerl`), for parsing and generation
- facilities to handle content to the web, to HTTP, etc. (`web_utils.erl`) and to perform **REST calls** (`rest_utils.erl`), using built-in `httpc` and `http_client` (including JSON services mentioned below)
- 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); nothing as advanced as Ceylan-Mobile, though
- 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 addressed to the use of OTP, in `otp_utils.erl`, allowing notably to run OTP applications out of an OTP context (typically through our native build/run system rather than through rebar3 and even OTP releases)
- services about all kinds of **units** (`unit_utils.erl`); refer to the Management of Units section below for more information
- basic services for **trace emission** (a.k.a logging - not related to Erlang tracing), either directly through `trace_utils.erl`, or thanks to `trace_bridge.erl` - typically to rely on more advanced trace systems such as Ceylan-Traces; now compliant with newer OTP logger and Syslog protocol as defined in RFC 5424, collecting bother userland traces and VM-level logs
- very basic facilities for **applications** (not in the sense of OTP ones), in `app_facilities.{e,h}rl` with an example (`most_basic_example_app.erl`)
- a basic support for **user-defined preferences**; this service relies on our preferences module to make available any user preferences expressed in the ETF format, and typically defined in a `~/.ceylan-settings.etf` file (possibly a symlink pointing to an actual file in VCS)
- a bit of **locale management**, in `locale_utils.erl`
- minor services about the **monitoring of Erlang processes**, in `monitor_utils.erl` and their **registering** in naming services, in `naming_utils.erl`
- facilities to better **interface Erlang to other languages**, in `language_utils.erl` and `{python,java}_utils.erl`; nothing as advanced as Ceylan-Seaplus, though

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`, `safe_maybe/1` and `fallible/{1,2}` 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`
  - `safe_maybe(T)` becomes the type union `'nothing'|{'just',T}` so that T may include the `undefined` (atom) value
  - `fallible(T)` becomes ultimately the type union `{'ok',T}|{'error',term()}`, while `fallible(Tok, Terror)` becomes `{'ok',Tok}|{'error',Terror}`
- both in type specifications and actual code, `table/2`, the Myriad-specific associative table pseudo-type, is translated into an actual table type
- the `const_table` module allows to generate sharable, read-only associative tables probably in the most efficient way in Erlang (refer to the const table section for further information)
- the `cond_utils` services drive conditional code injection: based on the build-time tokens defined, their values can be used to perform compilation-time operations such as **if** (see in this module `if_debug/1`, `if_defined/{2,3}`, `if_set_to/{3,4}`), **switch** (see `switch_set_to/{2,3}`, possibly with a

default clause) or **assert** (`assert/{1,2,3}`); if useful, it should be fairly easy (infrastructure mostly ready) to transform the (currently constant) user-defined build tokens into mutable variables and to add for example compile-time assignments (`cond_utils:create_token(TOKEN, MAYBE_INITIAL_VALUE)`, `cond_utils:set_token_value(TOKEN, VALUE)` and `cond_utils:remove_token(TOKEN, MAYBE_INITIAL_VALUE)`) of these variables and loops (**for**, **while**, etc.) if not going for a Turing-complete language, if ever that made sense for some uses; see the Support for Code Injection for additional usage details regarding the supported primitives

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.

Spatial Support

Motivation

The purpose of this section is to describe the facilities offered by Myriad in terms of **spatial operations**, i.e. computations relating to linear algebra, notably for 2D, 3D and 4D support.

We introduced these elements mostly for convenience, to have them readily available in a simple, controllable form, in pure Erlang, easy to enrich and without involving extra dependencies.

One possible example of their use is when relying on modern OpenGL (version 3+), in which the direct matrix support has been dropped (the so-called immediate mode does not exist anymore, except in a compatibility context). So now the application has to compute such matrices (model-view, perspective, etc.) by itself (on the CPU), as inputs to its GLSL shaders. For that, applications may use dedicated libraries, such as, in C/C++, GLM (OpenGL Mathematics); the linear support of Myriad aims to provide, in Erlang, a relevant subset of these operations.

This support is not expected to be specifically complete, battle-tested or efficient. If looking for such traits, one may consider:

- the elements already available directly in Erlang, notably the gl module, providing for example primitives to load OpenGL matrices and operate on them (note that this mode of operation is deprecated since OpenGL 3.0)
- in the Erlang community: Wings3D, an open-source modeller whose sources of course implement many spatial operations, notably in e3d (see the related section in our HOWTO)
- integrating advanced, non-Erlang libraries such as ones for linear operations implementing the BLAS specification; using the C binding (*CBLAS interface*) of a renown implementation (optimised at length and making use of processor-specific extensions), such as LAPACK and making it available to Erlang typically thanks to either NIFs (most suitable approach here) or a C-node (possibly thanks to Ceylan-Seaplus) would certainly be an option - all the more relevant that a bulk of linear operations could be offset as a whole to it; some Erlang projects target similar objectives, like linalg or matrex; more generally the services implemented by a library such as GSL (the *GNU Scientific Library*) could, thanks to a third-party project, become available to Erlang programs

In order to check the functional services and the correctness of operations, we recommend the use of Scilab or GNU Octave; refer to our quick HOWTO for further details.

Conventions

Linear Conventions Dimensions are non-null (a zero dimension has little interest). Dimension 1 corresponds to scalar and is not special-cased (hence one shall preferably use directly scalars if able to determine that being in a single dimension context).

A linear-related **index** (ex: of a coordinate of a point, a vector or a matrix) starts at `1` (not 0), as by default all indices in Myriad.

Points are to be specified by the user as *tuples* (preferably to lists) whose coordinates are either integer ones (for example $P = \begin{pmatrix} 10 \\ 45 \end{pmatrix}$ translating to `P={10,45}`, typically for GUI-related processing of on-screen coordinates) or floating-point ones (`{0.0, -1.0, 0.0}` for a point in 3D space). This is the most natural term mapping, and their internal representation is an homogeneous tuple (i.e. whose elements are all of the corresponding type): either `integer_point/0` or `point/0`.

The vast majority of the linear operations can be carried by modules operating either on:

- **arbitrary** dimensions (as high as needed, and freely chosen by the user, at compile-time or runtime)
- **specialised** dimensions, namely 2D, 3D or 4D; their interest lies in efficiency (these specialised constructs are designed to induce less computing and smaller memory footprints) and in the definition of dimension-specific operators (ex: the cross-products in 3D)

Points can therefore be of arbitrary dimension (then they are taken in charge by the `point` module), or can be specialised for 2D, 3D or 4D (then they are taken in charge by the `point{2,3,4}` modules).

As for **vectors**, they are to be specified by the user as *lists* of floating-point coordinates (ex: $\vec{V} = \begin{bmatrix} 0.0 \\ -7.3 \\ 3.22 \end{bmatrix}$ translating to `V=[0.0, -7.3, 3.22]`); this directly corresponds to their internal representation, in order to better accommodate linear operations (being a special case of matrices).

So vectors also can be of arbitrary dimension (then they are taken in charge by the `vector` module), or can be specialised for 2D, 3D or 4D (then they are taken in charge by the `vector{2,3,4}` modules).

Points and vectors (of arbitrary dimension, or specialised) can be converted both ways, see `point*:{to,from}_vector/1` and `vector*:{to,from}_point/1`. As their types differ (tuple versus list), they can be unambiguously discriminated, which is useful for some operations[22].

The **matrices** handled here can be of any dimensions (they are often square), and their elements are floating-point coordinates as well.

Let's consider a $m \times n$ matrix (m rows, n columns):

$$M = \begin{bmatrix} a11 & a12 & ... & a1n \\ a21 & a22 & ... & a2n \\ ... & ... & ... & ... \\ am1 & am2 & ... & amn \end{bmatrix}$$

With Myriad, such a matrix may be expressed:

[22] For example when applying a 3D point or a 3D vector to a 4x4 matrix, their fourth `W` coordinate will be respectively considered as being equal to `1.0` or `0.0`, and a corresponding normalisation of the other coordinates will be done only for points.

- as one of arbitrary dimension (designated from now on as an "*arbitrary matrix*"), corresponding to the `matrix:matrix/0` type; internally such matrices are nested lists: a list of `m` rows, each being a list of `n` elements, hence defined in row-major order (not column-major one)
- if being square and of a well-known dimension among 2, 3 or 4 (special cases defined for convenience and performance), as a value belonging to the `matrix{2,3,4}/0` types (which are records like `#matrix4{}`, whose fields are named according to the matrix elements, such as `m41`); they are designated hereafter as "*specialised matrices*"
- in a symbolic way, such as `identity_4` (meaning the identity 4x4 matrix)
- for some dimensions (ex: 3D or 4D), extra representations exist (compact 4x4 matrices, made of a 3x3 matrix and a vector3, in the context of homogeneous operations)

Taking as an example a 2x2 matrix like:

$$M = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix}$$

it can be created as an arbitrary `matrix/0` with:

```
M1 = matrix:new([ [A11,A12], [A21,A22] ])
```

Alternatively it can be directly created as a specialised (presumably more efficient) 2x2 matrix `matrix2` with:

```
M2 = matrix2:new([ [A11,A12], [A21,A22] ])
% Or:
M3 = matrix2:from_coordinates(A11, A12, A21, A22)
% Or even:
M4 = matrix2:from_arbitrary(M1)
M5 = matrix:specialise(M1)
```

There is a priori little interest in "unspecialising" (i.e. going from specialised to arbitrary matrix) by having:

```
M6 = matrix:unspecialise(M2)
```

In practice the actual, internal terms corresponding to all these matrices would be:

```
% For arbitrary ones:
% (supposing that all Axy coordinates are already floats):
M1 = M6 = [ [A11,A12],
            [A21,A22] ]

% For specialised ones:
M2 = M3 = M4 = M5 = #matrix2{ m11=A11, m12=A12,
                              m21=A21, m22=A22 }
```

Finally, **quaternions** are also supported (see `quaternion.erl`). They can be defined from 4 numbers, or as a 3D rotation. They are stored as quadruplets of floats, and can be added, multiplied, negated, scaled, normalised, conjugated, inversed, etc., and may be represented either as

$$Q = \begin{vmatrix} A \\ B \\ C \\ D \end{vmatrix}$$

or as:

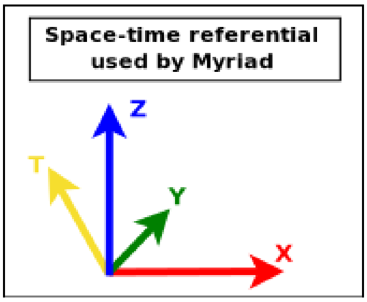
$$Q = A + B.\mathbf{i} + C.\mathbf{j} + D.\mathbf{k}$$

They notably provide a higher-level, more convenient counterpart to 3x3 rotation matrices (see `matrix3:rot_matrix3()`); both can be computed from a unit axis and an angle.

Note that:

- we call a container *type-homogeneous* iff all the coordinates that it gathers are all either integer or floating-point ones
- new instances (ex: of points, matrices, vectors, quaternions) may be:
  - either literally specified, with a term directly corresponding to their internal form
  - or based on a `new` operator (ex: `matrix:new/1`), in which case with a higher-level user-term (ex: a matrix with integer coordinates, in which case they will be automatically converted to floats)
- for clarity and in order to provide them with specified operations (like dot product), we preferred defining vectors as a separate type from the matrix one (even if a vector can be seen as a 1-column matrix)
- by default, for least surprise, coordinates are displayed in full, i.e. *not* rounded (refer to the `printout_{width,precision}` defines in `linear.hrl`)
- the procedure to check the validity of computations is the following:
  - first implement the arbitrary version
  - validate it, by composing internal operations and by comparison with a tool like Scilab/Octave
  - implement the specialised versions
  - validate them against the arbitrary version
- the most common operations are defined for each datatype: creating, modifying, comparing, displaying and, whenever appropriate: adding, subtracting, scaling, multiplying, rotating, measuring, transposing, reversing, etc.
- operations are not implemented defensively, in the sense that a base runtime error will be triggered if a type or a size does not match, rather than being tested explicitly (anyway generally no useful extra context could then be specifically reported)

- additional runtime checks (ex: to check whether parameters expected to be unit vectors are normalised indeed) can nevertheless be enabled by setting the `myriad_check_linear` flag (refer to `GNUmakevars.inc`)
- for homogeneous coordinates: any implicit homogeneous *w* coordinate is `1.0`
- most operations here involve floating-point coordinates, rather than integer ones; as an Erlang's `float()` is a double-precision one, it requires more resources (CPU and memory footprint) than a basic, single-precision one; for applications not requiring extra precision, maybe the Erlang VM could be compiled in order to rely on single-precision floats instead



Geometric Conventions For **space** coordinates, three axes are defined for a global referential:

- abscissa: X axis (in red, `#FF0000`)
- ordinate: Y axis (in green, `#008000`)
- depth: Z axis (in blue, `#0000FF`)

By default, we consider right-handed Cartesian coordinate systems (like OpenGL; unlike DirectX or Vulkan), and we rely on "Z-up" conventions (the Z axis being vertical and designating altitudes), like modelling software such as Blender[23].

In 2D, typically for on-screen coordinates (ex: when drawing in a canvas), the corresponding projected referential applies, based on the X and Y axes, the origin being in the top-left corner, and all Z coordinates being zero[24]:

For each of the spatial dimensions of interest, generally `1.0` corresponds to 1 meter, otherwise to 1 light-second (i.e. roughly 300 000 km[25]).

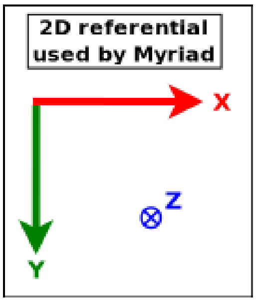
For **all angles**, the default unit is the radian (2π radians is equal to 360 degrees), and the positive rotation is counterclockwise.

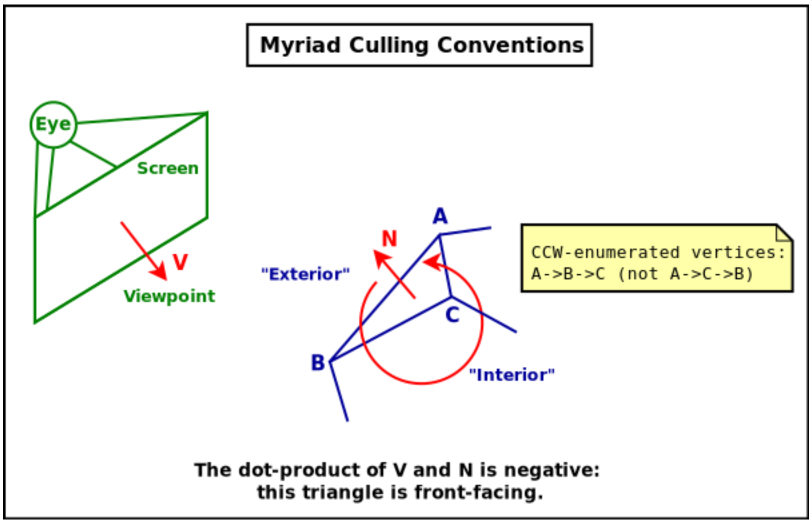
For **face culling**, front-facing is determined by relying on a counter-clockwise order winding order of triangles (like default OpenGL's GL_CCW):

[23] Unlike many games, for which the Y axis is up, Z being the depth, perpendicular to the screen. Anyway a simple camera transformation is enough to switch conventions.

[24] This 2D referential corresponds to the base space-time one, when the viewpoint is located in the negative Z axis and looks at the origin.
One may also refer to `gui_opengl:enter_2d_mode/1` to apply these conventions.

[25] Then for more human-sized distances, a scale of one light-nanosecond (10^-9 second) might be more convenient, as it corresponds almost to 30 cm.





Indeed a triangle enumerating its vertices in counter-clockwise order (`A->B->C`) would define a normal vector $\vec{N} = \overrightarrow{AB} \times \overrightarrow{BC}$ pointing towards the outside of a body comprising that triangle.

If $\vec{V} \cdot \vec{N}$ (i.e. the dot-product of the view direction vector and of this outward vector product) is:

- strictly negative: then the face is front-facing
- positive: then the face is rear-facing

Said otherwise, front-facing polygons are the ones whose signed area (determinant) is strictly positive; see also: `polygon:{get_area,get_signed_area}/1`.

A fourth coordinate besides X, Y and Z could be used, as an extra axis (in yellow, `#F6DE2D`):

- either for **homogeneous** coordinates, in which case it will be considered to be spatial as well, with the same unit as the three first ones, and preferably designated as `W`
- or for **time** coordinates, with a single axis defined for a global referential: the `T` one, for which `1.0` corresponds to 1 second

We consider that **clip space** ranges in `[-1.0, 1.0]` (like OpenGL conventions; rather than for example `[0.0, 1.0]`, which are the D3D ones).

Related Services Elements of interest can be:

- some support of polygons, in `polygon.erl` - a basic support for (2D) bounding surfaces (including rectangles, "lazy" circles and *Minimal Enclosing Circles* based on convex hulls; see `bounding_surface.{hrl,erl}`) and corresponding (3D) bounding volumes (including right cuboids and spheres; see `bounding_volume.{hrl,erl}`)
- elements in order to import/export 3D scenes thanks to the glTF file format

Possible Enhancements In the future, the most usual spatial types such as `matrix` and `vector` may be shortened in Myriad-based code as respectively `m` and `v`, the Myriad parse transform being then in charge of expanding accordingly (ex: a in-code shorthand `m3:new/0` becoming `matrix3:new/0` to the eyes of the compiler).

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[26] 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`][27] (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)`.

[26] 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`.

[27] 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)[28], 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:

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

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

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 Backends

To perform SQL operations, a corresponding software solution must be available.

Two SQL database backends have been preferred here: **SQLite 3** for lighter needs, and **PostgreSQL** for heavier ones, notably whenever having to connect to a third-party instance thereof.

SQLite 3

SQLite Basics The SQLite 3 library 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 backend as it is:

```
$ 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.

SQLite Binding This database system can be directly accessed (as a client) 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 backend 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.

PostgreSQL

PostgreSQL Basics PostgreSQL is a well known, free and open-source (client/server) *Relational Database Management System* (RDBMS) emphasizing extensibility and SQL compliance. It is designed to handle a range of workloads, from single machines to larger services with many concurrent users.

This object-relational database can be enriched to support other datatypes, such as geographic objects with PostGIS.

It can be installed on Debian thanks to the `postgresql-client` package (and the database itself with the `postgresql` one), or `postgresql` on Arch Linux.

PostgreSQL Binding This database system can be directly accessed (as a client) thanks to either the PostgreSQL command-line client (`psql`; for a simpler, more limited approach) or to an Erlang binding, the one that we retained being epgsql (for a more efficient, in-depth access).

By default we consider that this binding has been installed in `~/Software/epgsql`. The `POSTGRESQL_BASE` variable in `myriad/GNUmakevars.inc` can be set to match any other install path.

Recommended installation process:

```
$ mkdir -p ~/Software && cd $_
$ git clone https://github.com/epgsql/epgsql.git
$ cd epgsql && make all && ln -s ./_build/default/lib/epgsql/ebin
```

Myriad SQL Support

To enable this support, once the corresponding backend and binding (see Database Backends) 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` backend not correctly installed (ex: `SQLITE3_BASE` not pointing to a right path in `myriad/GNUmakevars.inc`)

`Myriad` Main Conventions

We list here the conventions of all sorts that the Myriad code (base or contributed one) - and also code in the software stack based on it - shall obey.

Text Conventions

The purpose here is to ensure a sufficient **code homogeneity**; for example in all source files are in such a "canonical form", analysing their differences (`diff`) is made simpler.

Any text editor can be used, provided that it saves source files with the UNIX, not DOS, conventions (i.e. lines terminating by the LF character, not by the CRLF characters).

The use of syntax highlighting is encouraged.

Recommended text editors are:

- Emacs
- Visual Studio Code (a.k.a. Vscode)
- ErlIDE (based on Eclipse)
- Vim, IntelliJ, Gedit, Nedit, etc.

The main editors integrate the *Language Server Protocol* (also known as LSP), refer to the LSP section for more details (Flycheck can be another option).

Source files should be formatted for a 80-character width: no character should be present after the 79th column of a line.

Except in very specific cases, only ASCII code should be used (ex: no accentuated characters).

Tabulations should be preferred to series of spaces, and the text should be formatted according to 4-character tabulations.

All redundant whitespaces should be removed, preferably automatically (see the Emacs `whitespace-cleanup` command). This is why, with the emacs settings that we recommend, pressing the F8 key removes for example the yellow areas in the current buffer by replacing any series of four spaces by a corresponding tabulation.

We would prefer that all files (especially source ones; including the contributed ones) are "whitespace-clean" before being committed. As mentioned, such a transformation can be done directly from Emacs. If using another editor, please ensure that the fix-whitespaces.sh script has been run on the target sources (possibly automatically thanks to a VCS hook) *before* committing them; the fix-whitespaces-in-tree.sh script may be also used, in order to perform a bulk transformation.

All elements of documentation should be written in English, possibly translated to other languages. Spell-checking is recommended.

Coding Practices

In terms of coding style, we would like that the sources remain as uniform as possible, regarding naming, spacing, code/comments/blank line ratios.

We would like that, roughly and on average, the same ratio applies for blank lines, comments and lines of code.

For that one may use the either directly `make stats` from the root of the layer or the make-code-stats.sh script.

For example:

```
In the Erlang source code found from XXX/ceylan/myriad, we have:
+ 208 source files (*.erl), 36 header files (*.hrl)
+ a grand total of 118666 lines:
  - 35959 of which (30.3%) are blank lines
  - 37187 of which (31.3%) are comments
  - 45520 of which (38.3%) are code
```

The most obvious conventions are:

- the **settings of the build chain** should be used (ex: with regard to compiler flags) and adapted/completed if needed; the (possibly-specialised) `GNUmakesettings.inc`, `GNUmakerules.inc` and `GNUmakevars.inc` files should be relied upon
- **no warning should be tolerated**; anyway our build chain treats warnings as (blocking) errors
- **test cases** should be developed alongside most if not all modules; ex: if developing `src/foobar.erl`, then probably the `test/foobar_test.erl` testing code should be developed, after or, even preferably, before the implementation of the tested code; test success should be evaluated automatically, by the code (ex: thanks to pattern matching), not by the person running the test (ex: who would have to compare visually the actual results with the expected ones); in some cases, only **integrated tests** can be devised in practice; tests should be gathered in **test suites**, that should be runnable automatically (`make test`, recursively through child directories) and fail loudly (and in a blocking manner) at the first error met
- **multiple levels of quality documentation** should be made available to the code user, and probably be written in parallel to the code; there are at least three documentation levels:
  - lower-level documentation: code should always be **densely commented**, with documentation headers added to all functions, inlined comments (not paraphrasing the code) and self-describing symbols: function names, variable names (ex: `RegisteredState = ...` to be preferred to `NewState = ...`), etc.; more generally all names shall be long enough to be descriptive (clarity preferred over compactness); type specifications also pertain to this low-level documentation effort
  - higher-level **design and/or implementation notes**: they should be available as a set of paragraphs in each source file, before the function definitions, to describe the context and constraints, and help understanding how the features are implemented, and why
  - high-level **developer and user documentation** should be made available, targeting at least HTML and PDF outputs, possibly offering a wiki access as well

- more generally, **comments** should be clear and precise, numerous, rich and complete (overall, in terms of line counts, we target roughly 1/3 of code, 1/3 of blank lines and 1/3 of comments); all comments shall be written in UK English, start with a single `%` and be properly word-wrapped (use `meta-q` with our Emacs settings to take care of that automatically)

- **indentation** should respect, as already explained, the 80-character width and 4-space tabulation; however the default built-in Erlang indentation mode of `emacs` can hardly be used for that, as it leads to huge width offsets (we may in the future provide a variation of the `elisp` code for the emacs indentation rules)

- **spacing homogeneity** across source files should be enforced; for example three blank lines should exist between two function definitions, one between the clauses of any given function (possibly two in case of longer clauses); for the sake of "visual parsing", arguments should be separated by spaces (ex: `f( X ) -> ...`, not `f(X) -> ...`), especially if they are a bit complex (`f( A={U,V}, B, _C ) -> ...`, not `f(A={U,V},B,_C) -> ...` or the usual `f(A={U,V}, B, _C) -> ...`)

- for **type-related conventions**, at least all exported functions shall have a `-spec` declaration; if an actual type is referenced more than once (notably in a given module), a specific user-defined type shall be defined; types shall be defined in "semantic" terms rather than on technical ones (ex: `-type temperature() :: ...` than `float()`; developers may refer to, or enrich, `myriad/src/utils/unit_utils.erl` for that)

- the **latest stable version of Erlang** should be used, preferably built thanks to our `myriad/conf/install-erlang.sh` script

- the official *Programming Rules and Conventions* should be enforced, as defined here (possibly a dead link now; one may try this mirror instead)

- the function definitions shall follow **the same order** as the one of their exports

- helper functions **shall preferably be identified as such**, with an `(helper)` comment

- if an helper function is specific to an exported function, it shall be defined just after this function; otherwise it should be defined in the **helper section**, placed just after the definition of the exported functions

- defining distinct (non-overlapping), explicit (with a clear-enough name), numerous (statically-defined) **atoms** is cheap; each atom found in the sources is generally to be involved in at least one type definition

- the use of `case ... of ... end` should be preferred to the use of `if` (never used in our code base)

- we also prefer that the various patterns of a case are indented with exactly one tabulation, and that the closing `end` lies as much as possible on the left (ex: if having specified `MyVar = case ... end`, then `end` should begin at the same column as `MyVar`); the same applies to `try ... catch ... end` clauses

- when a term is ignored, instead of using simply `_`, one should define a **named mute variable** in order to provide more information about this term (ex: `_TimeManagerPid`); one should then to accidental matching of such names (now a warning is emitted)
- some conventional variable names are, and may be, extensively used: `Res` for result, `H` and `T` for respectively the head and tail of a list on which we recursively iterate
- indices shall, as much as possible, start at index `1` (rather than 0); this is a general Erlang convention (for lists, like with `lists:nth/2`, for tuples, etc. - unlike arrays, though); see `basic_utils:positive_index/0`
- when needing an **associative table**, use the `table` pseudo-module; a key/value pair shall be designated as a table *entry* (ex: variable named as `RoadEntry`)
- regarding the in-code management of **text**:
  - if a text is to be rather static (constant) and/or if it is to be exchanged between processes, then it should be a UTF8 `binary`, and its type shall be declared as `text_utils:bin_string()`
  - other, a plain string (`text_utils:ustring()`) shall be used
- when defining a non-trivial datastructure, a **record** shall be used (rather than, say, a mere ad-hoc tuple or a map of undocumented structure...), a corresponding **type** should be then defined (ex: a `foobar` record leading to a `foobar()` type), and a **function to describe it** as text shall be provided (ex: `-spec foobar_to_string(foobar()) -> text_utils:string()`)
  - **mute variables** should be used as well to document actual parameters; for example `f(3,7,10)` could preferably be written as a clearer `f(_Min=3,_Max=7,_Deviation=10)`

Note

Mute variables are however actually bound, thus if for example there is in the same scope `_Min=3` and later `_Min=4`, then a badmatch will be triggered at runtime; therefore names of mute variables should be generally kept unique in a given scope.

- as opposed to records, types shall never defined in header files (`*.hrl`): a given type shall be defined once, as a reference, and exported by its module; other modules may then just refer to it
- type shorthands may be defined; for example, if using repeatedly within a module `text_utils:ustring()`, a local, non-exported type shorthand (`-type ustring() :: text_utils:ustring()`) may be defined so that all other uses of this type become simply `ustring()` in this module

As not all typos may be detected at compilation-time (ex: wrong spelling for a module), we recommend, for source code, the use of additional static checkers, as discussed in the type-checking section.

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 the `EXECUTION_TARGET` variable 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").
```

See also the (different) Wings3D coding guidelines, that are interesting in their own right.

Tooling Conventions

Erlang LS

The Language Server Protocol (also known as LSP) may be used by one's editor of choice in order to provide various services facilitating the developments in various languages, including Erlang.

For that Erlang LS should be used. We then recommend to rely on our erlang_ls.config configuration file, which may be installed that way:

```
$ mkdir -p ~/.config/erlang_ls && cd ~/.config/erlang_ls/
$ ln -sf ${CEYLAN_MYRIAD}/conf/erlang_ls.config
```

As for the installation of Erlang LS itself, we rely on:

```
$ mkdir -p ~/Software && cd ~/Software
$ git clone https://github.com/erlang-ls/erlang_ls
$ cd erlang_ls/
$ make
$ mkdir bin && cd bin
$ ln -s ../_build/default/bin/erlang_ls
```

Then one would just have to ensure that `~/Software/erlang_ls/bin` is indeed in one's PATH.

Note that not all bells and whistles of LSP may be retained, knowing that at least some of them are confused by various elements, especially when applied to code that is parse-transformed; as a result we did find LS features much useful.

The Emacs configuration that we use (see the corresponding init.el) attempts to find some sweet spot in this matter.

Another option is to use `ctags` to generate Emacs' compliant tags (see the `generate-tags` make target); we did not find this solution very satisfactory either.

For Documentation Generation

Refer now to our dedicated HOW-TO.

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), `Acc` designating accumulators
- as there is much list-based recursion, a variable named `H` means `Head` and `T` means `Tail` (as in `[Head|Tail]`)
- the string format specifier `~s` shall never be used; its Unicode-aware counterpart `~ts` must be used instead; similarly, for string operations, `list_to_binary/1` and `binary_to_list/1` must no be used either; prefer anyway the primitives in `text_utils`

Myriad-related Troubleshooting

Header/Module 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).

Third-Party Dependencies

Let's suppose we have an application named `Foo` that relies on Myriad.

`Foo` may define additional dependencies, which may be:

- either mandatory or optional
- needed starting from build-time (ex: if relying on their headers and/or modules - including parse-transforms), or only at runtime

For a given **optional** dependency (ex: regarding JSON), a USE make variable is to be defined in the layer that introduced this dependency (ex: `USE_JSON`, introduced by Myriad, therefore to be listed in its `GNUmakevars.inc`). This variable allows to have our native build system register the associated additional include and ebin directories.

The first step to enable such a dependency (ex: the JSON support) is to set its USE variable to `true` (ex: `USE_JSON = true`), as it is expected to be disabled by default. Depending on the service at hand, a specific backend may have also to be selected (ex: either `USE_JSX = true` or `USE_JIFFY = true` to select a suitable JSON parser).

Finally, some supports may depend on others (ex: enabling `USE_REST` will enable `USE_JSON` in turn).

Runtime-only Third-Party Dependencies

The dependencies discussed here have to be found only when *executing* one's application; they can be installed:

- either manually, in which case the location of their `ebin` directory (typically an absolute path then) shall be specified in the code path (see, in `GNUmakevars.inc`, the `JSX_SOFTWARE_BASE` make variable for an example)
- or thanks to rebar, in which case they shall obey the same rules as the Build-time Third-Party Dependencies discussed below

Build-time Third-Party Dependencies

Myriad does not have such dependencies, but layers above in the software stack (like a layer that would be named `Foo`) may.

To have such dependencies (ex: let's suppose that the `jsx` JSON parser defined header files that one wants to include) *installed* as well when building one's project (ex: `Foo`), one may rely on rebar, and list them in the project's `foo/conf/rebar.config.template` file (ex: `{deps, [bar, jsx]}.`) from which the actual `rebar.config` is to be generated (use the `make set-rebar-conf` target for that).

The actual compilation will be done by our native build system in all cases, either directly (when running `make all`) or when using `rebar compile` (rebar hooks will then ensure that in practice the application is compiled with our native rules anyway). Therefore appropriate make variables (ex: `JSX_REBAR_BASE`, in `myriad/GNUmakevars.inc`) shall be defined so that the corresponding BEAM files installed through rebar can be found in this native context as well (through the `BEAM_DIRS` make variable).

Finally, such dependencies may or may not be listed in the `deps` entry of the `conf/foo.app.src` file[29], depending on whether they are optional or not.

Myriad-level Third-Party Dependencies

Myriad as such has no mandatory dependency (except Erlang itself of course), but *optional* ones may be enabled, for:

- a basic JSON support (see our json_utils module), thanks to a suitable actual JSON parser, either jsx or jiffy; note that the detection and use of these parsers are done transparently at runtime, hence none of them is a declared dependency of Myriad, which will adapt to any choice made by the overall application that includes both Myriad and one of such parsers (provided, as mentioned above, that the proper `USE_*` make variables are set)
- a first-level support of the HDF5 file format (see our hdf5_support module), based on - and thus requiring - the enhanced fork that we made of erlhdf5
- Python (see our python_utils module), thanks to erlport
- SQLite (see our sql_support module), thanks to the SQLite 3 Erlang binding that we retained, erlang-sqlite3

As an example, let's suppose that we need a JSON support and that we want to rely on the `jsx` parser (our default choice) for that.

If applying our conventions, supposing that Erlang and Rebar3 are already installed (otherwise refer to the getting Erlang and getting Rebar3 sections), `jsx` may be installed with:

[29] After having edited this file, run `make rebar3-create-app-file` afterwards in order to have the three other versions of it properly generated (namely `./_build/lib/foo/ebin/foo.app`, `ebin/foo.app` and `src/foo.app.src`).

```
$ mkdir -p ~/Software/jsx
$ cd ~/Software/jsx
$ git clone https://github.com/talentdeficit/jsx.git
$ ln -s jsx jsx-current-install
$ cd jsx/
$ rebar3 compile && rebar3 eunit
```

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 (generally) references to the `map_hashtable` module. See table transformations for more information.

Enabling the Interconnection of Erlang nodes

This is not a Myriad gotcha per se, but rather an Erlang one.

Way too often, for obscure reasons Erlang nodes fail to connect to each other (especially with long names), and little to no information is available to diagnose the issue.

Safety Measures

In order to maximise the chances that nodes are able to `net_adm:ping/1` successfully each other:

- at least for testing, run VMs spawned with preferably the same **version** of Erlang
- ensure that they rely on the same **EPMD** (TCP) port (default Erlang one is `4369`, while Myriad default one is `4506`); check for example that all launched nodes of interest can be seen with: `epmd -port 4506 -names`
- ensure that both nodes use either short names (start a new node with `-sname`) or long names (start a new node with `-name`)
- check that they use the same **cookie**, either from the start (use the `-setcookie MY_COOKIE` command-line option) or after having changed it after the VM was launched (use `erlang:set_cookie/1` for that / check with `erlang:get_cookie/0`)
- ensure that no **firewall** gets in the way; one may take inspiration for example from our iptables.rules-FullDisabling.sh script
- finally check that the local **DNS resolution** complies with the surprisingly picky constraints demanded by the Erlang VM

For this last point, `/etc/hosts` is often the scene of the disaster. If your hostname is `hurricane` and your domain is `foobar.org`, then a line like the following one is known to work (whereas many variations of it may be deemed "incorrrect"):

```
127.0.0.1  hurricane.foobar.org hurricane localhost.localdomain localhost
```

provided of course that, still in that file, you have not also a declaration such as:

```
192.168.0.5 hurricane.foobar.org hurricane
```

(setting one's IP shall better be done in one's profile in `/etc/netctl`, right?)

Testing & Troubleshooting

In order to **quick-check** whether long-name connectivity is available and to rule out the most obvious culprits, open two terminals.

In the first:

```
# Check (with root permissions) that the firewall rules are safe; for example:
$ iptables -nL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

# Just to be on the safer side for this test:
$ killall beam.smp epmd

# Then launch the target first node:
$ ERL_EPMD_PORT=4032 erl -name n1 -setcookie aa
Erlang/OTP 23 [erts-11.1.4] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1]

Eshell V11.1.4  (abort with ^G)
(n1@hurricane.foobar.org)1>
```

In the second terminal, try to find the previous node:

```
$ ERL_EPMD_PORT=4032 erl -name n2 -setcookie aa
Erlang/OTP 23 [erts-11.1.4] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1]

Eshell V11.1.4  (abort with ^G)
(n2@hurricane.foobar.org)1> net_adm:ping('n1@hurricane.foobar.org').
pong
```

If you see `pang` here, run to the nearest altar and make a sacrifice to any Distribution God you may believe in (Norse ones being presumably the most effective here), and apply the hints listed in the Enabling the Interconnection of Erlang nodes section.

To better troubleshoot, one may also spawn two test nodes and see whether they are able to ping each other.

Using the Erlang Shell for Debugging

It may be convenient to run an Erlang shell in order to investigate and fix issues.

One may execute `make shell` to launch a shell that is parameterised so that all modules of all layers (hence having Myriad from Myriad) are in its code path.

The built-in shell commands are then very convenient, notably:

- `v(-1)` to get the *result* of the last command
- less relevant in a Myriad context: `c(my_module)` to compile (if possible with default settings - thus notably with no parse transform involved) and (re)load the specified module
- `l(my_module)` to (re)load the specified module; useful when it has to be recompiled by Myriad (typically thanks to a `make` issued in another terminal)

Do not mix up this last command with `rl(XXX)`, which does not perform a module reload but prints a record definition (and will not complain if given an unrelated module name, thus not reloading anything...).

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, the Myriad Text Conventions and Coding Practices are respected.

Thanks in advance, and have fun with Ceylan-Myriad!