Sim-Diasca Developer Guide

Contact: olivier (dot) boudeville (at) edf (dot) fr

Author: Olivier Boudeville

Creation Date: February 2009

Lastly updated: Thursday, September 28, 2023

Version: 2.4.6

Status: Stable

Website: http://sim-diasca.com

Dedication: To the Sim-Diasca developers

Abstract: This document summarizes the main conventions that should be respected when contributing code to Sim-Diasca and/or making use of Sim-Diasca.

Table of Contents

Note

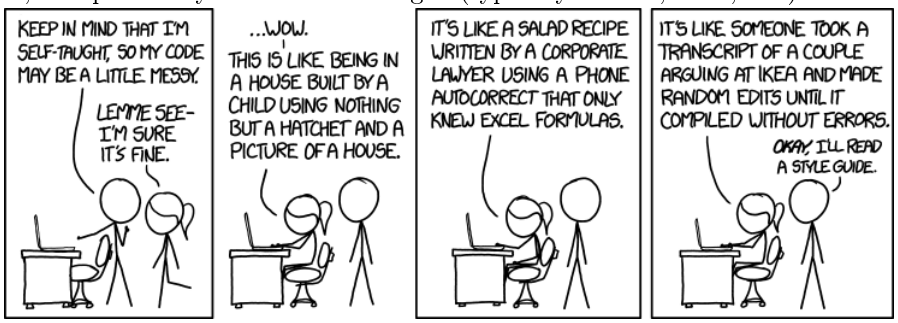
This document intends to gather information mostly aimed at Sim-Diasca *maintainers* or *contributors* - not users. Most people should read the *Sim-Diasca Technical Manual* first, and possibly the *Sim-Diasca Modeller Guide* as well.
However we require that the in-house authors of any simulation element making use of Sim-Diasca's services (ex: models, simulation cases, etc.) respect the conventions presented in the current document, for the sake of the clarity and homogeneity of the code base.
As we believe that these conventions may benefit to third-party users, we share them as well.

1 Sim-Diasca Code Conventions

1.1 Foreword

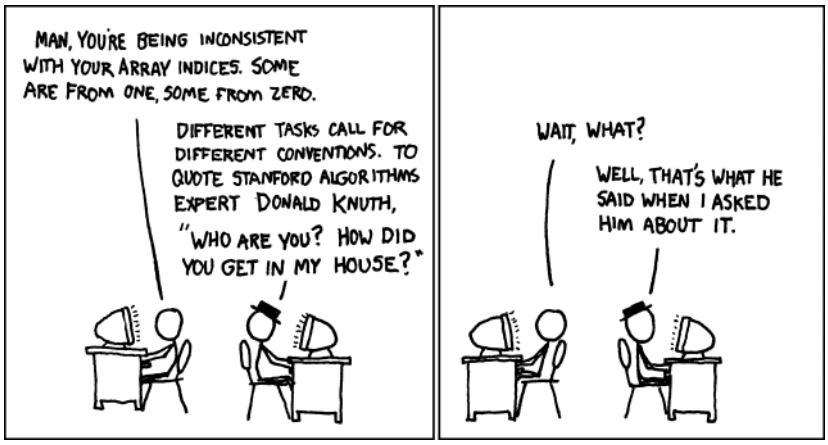
At all levels of the technical architecture, we tried to enforce a few conventions, that are detailed below.

Some of them are necessary, others are mere good practices, a few are arbitrary (notably in terms of style), but we believe that, for the sake of clarity and homogeneity, all of them should be respected in the code of the Sim-Diasca stack, and preferably also in code using it (typically models, tools, etc.).



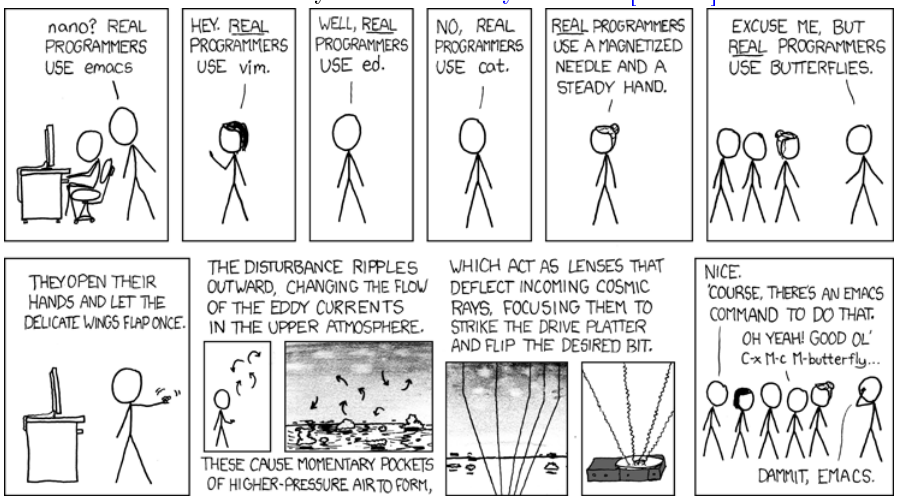
(see the credits section about the comic strips)

We believe also that these conventions have been fairly well enforced in our own Sim-Diasca code base (which thus might be used as an example thereof). Please tell us if you do not think so, or if you identified interesting other conventions that could be listed here and applied.



1.2 Text Conventions

We now recommend to directly stick to the Myriad ones [mirror].



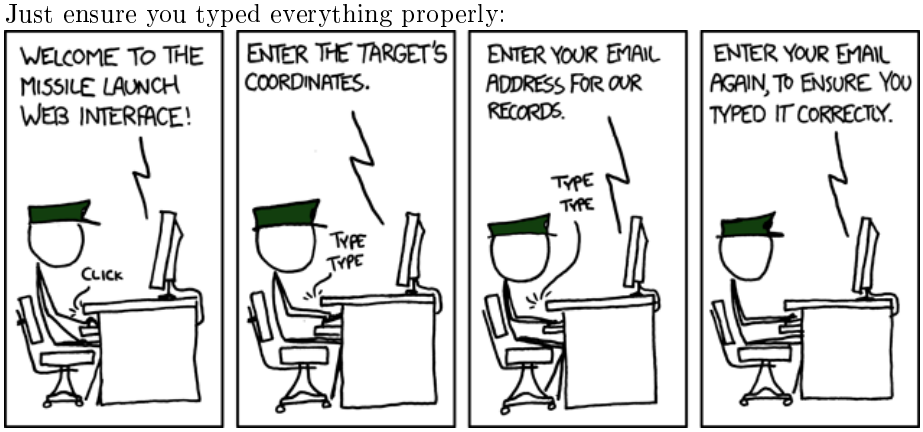
class_TimeManager.erl: /home/boudevil/Projects/Sim-Diasca/sources/Sim-Diasca/sim-diasca/src/core

File Edit Options Buffers Tools Erlang Help

```
1066
1067 % Sets the initial tick to be used by the simulation, based on the specified
1068 % time and date, expressed in virtual time.
1069 %
1070 % Must not be called while the simulation is running.
1071 %
1072 % (oneway)
1073 %
1074 -spec setInitialSimulationDate( wooper_state(), unit_utils:date(),
1075                                  unit_utils:time() ) -> oneway_return().
1076 setInitialSimulationDate( State, Date, Time ) ->
1077
1078     false = ?getAttr(started),
1079
1080     NewInitialTick = timestamp_to_ticks( { Date, Time }, State ),
1081
1082     ?info_fmt( "New initial simulation tick is ~B.", [ NewInitialTick ] ),
1083
1084     ?wooper_return_state_only( setAttribute( State, initial_tick,
1085                                              NewInitialTick ) ).
1086
1087
1088
1089 % Returns the final (absolute) tick for that simulation, or the 'undefined' atom
1090 % if no final tick was defined.
1091 %
1092 % (const request)
1093 %
1094 -spec getFinalTick( wooper_state() ) ->
1095                           request_return( basic_utils:maybe( tick() ) ).
1096 getFinalTick( State ) ->
1097
1098     Res = case ?getAttr(stop_tick_offset) of
1099
1100               undefined ->
1101                   undefined;
1102
1103               StopOffset ->
1104                   ?getAttr(initial_tick) + StopOffset
1105
1106     end,
1107
1108     ?wooper_return_state_result( State, Res ).
1109
1110
1111
```

--:--- class_TimeManager.erl 18% (1087,0) Git-sim-diasca-2.1.2 (Erlang WS)----------

Just ensure you typed everything properly:



1.3 General View of the Software Stack

We see Sim-Diasca as a stack of layers, so that a given layer only depends on the ones below it, and never from the ones above.

Top-to-bottom, we have:

Layer Name	**Role**
Sim-Diasca	This simulation engine
Ceylan-Traces	The distributed trace system
Ceylan-WOOPER	The object-oriented layer
Ceylan-Myriad	The base library offering general-purpose services
Erlang	The base language and environment

Thus, there is not upward dependency, for example WOOPER depends on Myriad and Erlang but not on Traces or on Sim-Diasca.

The other way round, bottom-up one can see:

- Erlang, which provides the way of defining and running concurrently a large number of processes
- Myriad, which gathers all common services that are needed, in terms of data-structures, lower-level constructs, most frequent processings, etc.
- WOOPER, which transforms Erlang processes into instances of classes with multiple inheritance, still running concurrently
- Traces, which allows each distributed instance to send appropriate traces
- Sim-Diasca, which transforms a distributed object-oriented application into a simulation

On top of that stack, which provides the simulation engine, there are at least:

- a set of models integrated into the Sim-Diasca framework
- a simulation case, which makes use of these models and organises them in the context of a scenario to be simulated, a virtual experiment

The simulation engine being itself absolutely generic as long as discrete-time simulations are involved, it may be convenient to define, on top of Sim-Diasca and below the actual models themselves, a domain-specific layer that specialises the framework in order to ease the development of models.

For example, a telecom-centric simulation could define building blocks like service queues, and mother classes like communicating nodes, network interfaces, packet loss models, etc.

1.4 Erlang Conventions

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 now our build chain treats warnings as (blocking) errors
- **test cases** should be developed alongside most if not all modules[1]; ex: if developing `class_X.erl`, then probably the `class_X_test.erl` testing code should be developed, after or, 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`) 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 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 help understanding how the features are implemented, and why
  - high-level **developer and user documentation** should be made available, targeting at least a PDF output, 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)

- **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 (the `elisp` code for the emacs indentation will be modified for our need, in the future); the Sim-Diasca conventional indentation should be enforced, preferably automatically (ex: thanks to `emacs`)

- **spacing homogeneity** across Sim-Diasca 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), 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) -> ...`)

- see the Using Type Specifications With Sim-Diasca section 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 function definitions shall follow **the same order** as the one of their exports

- helper functions **shall be identified as such**, with an `(helper)` comment; the same stands for all other kinds of functions mentioned in next sections

- 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; they are 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
- 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
- when needing an **associative table**, use the `table` pseudo-module; if needing to store such an instance in an attribute, its name shall be suffixed with `_table` (ex: `road_table`); a key/value pair shall be designated as a table *entry* (ex: variable named as `RoadEntry`)
- regarding **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 `binary`, and its type shall be declared as `text_utils:bin_string()`
  - other, a plain string (`string()`) shall be used
- when defining a non-trivial datastructure, a **record** shall be used (rather than, say, a mere ad-hoc tuple), 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()) -> 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.

[1] In terms of directories, the source of modules (`*.erl`) shall be in `src`, the includes (`*.hrl`) in `include`, the tests (`*_test.erl`) in `test` - in each case, either directly in the specified directory, or at any depth in nested subdirectories.

For the sake of clarity, we try to avoid too compact code, and code too poorly understandable for everyone but its original creator. Thus we want to enforce a minimum ratio of blank lines and comments.

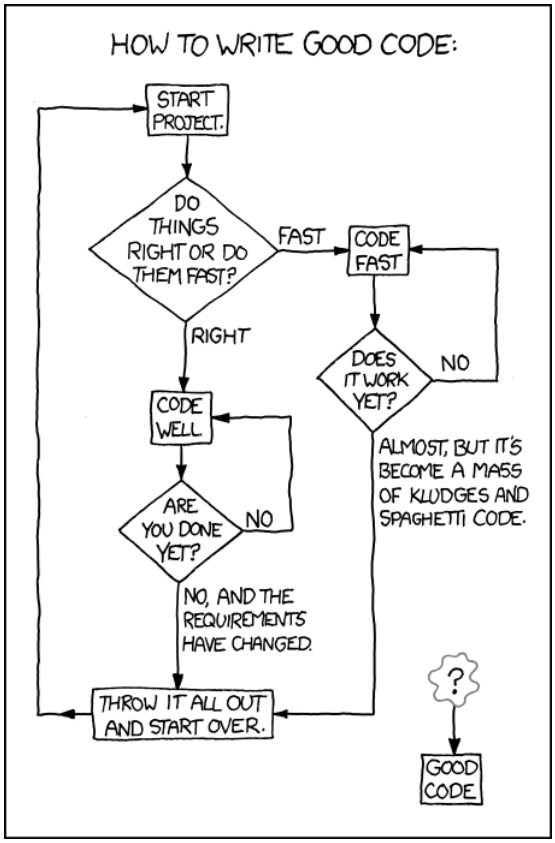
For example, as of May 2017, we have for the Sim-Diasca stack (i.e. from `myriad` to `sim-diasca`):

- 326 source files (`*.erl`), 86 header files (`*.hrl`)
- a grand total of 178980 lines: - 57814 of which (32.3%) are blank lines - 56548 of which (31.5%) are comments - 64618 of which (36.1%) are code

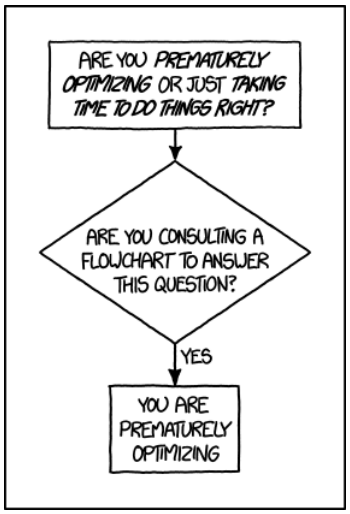
These information can be obtained by running `make stats` from the root of a Sim-Diasca install.

Other recommended good practices are:

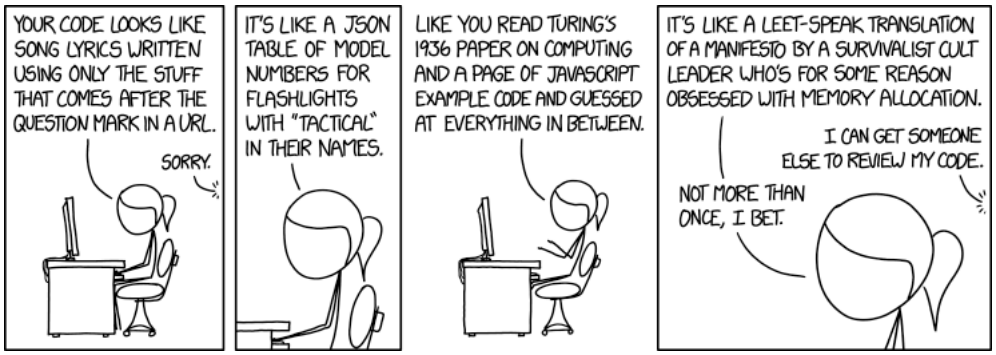
- **peer review**: before committing code, or before issuing a release, it should be reviewed by someone who is not the one who introduced the corresponding changes
- write **type specifications** and run regularly **Dialyzer**



Another piece of advice we maybe should apply more frequently:



Of course we cannot stress enough that securing a sufficient code quality is essential for the other developers to come, and also often even for one's future self; for that reason we recommend pair-programming, or at least the aforementioned review before new sources are incorporated in the code base:



1.5 Myriad Conventions

The general goal is to collect recurring generic lower-level patterns and solutions in that layer.

When an helper mechanism is already available in `Myriad`, it should be used instead of being defined multiple times in the software stack.

Reciprocally, when a well-defined generic sequence of instructions is used more than once, it should be integrated (commented and tested) in that `Myriad` layer.

Main services are:

- support of generic data-structures, like hashtables, option lists, etc. (in `data-management`)
- some helpers for GUI programming (in `user-interface`)
- support of some math-related operations, mostly linear (in `maths`)
- various helpers, for system-related operations, text management, network operations, executable support, unit management, etc. (in `utils`)

For further information, please refer to the **Technical Manual of the Myriad Layer**[2].

1.6 WOOPER Conventions

One should respect the WOOPER conventions (please refer to the WOOPER documentation).

For example, `oneway`, `request` or `helper function` (possibly with qualifiers like `const`) should be specified in each method header.

WOOPER type conventions shall be used as well, for example:

```
-spec getFoo( wooper:state() ) -> request_return( foo() ).
```

Method names should be spelled in `CamelCase` (ex: `getColorOf`, not `get_color_of`).

Variables bound to a WOOPER state shall have their name suffixed with `State`; ex: `NewState`, `UpdatedState`, etc.

To better discriminate between methods and functions (ex: helpers):

- the latter shall have their name spelled in `snake_case` (ex: `update_table`, not `updateTable`)
- should an helper function have among its parameters the state of an instance (type: `wooper:state()`; typically either to access directly to attributes or to use trace primitives), this parameter should be listed last (ex: `update_table(X,Table,State)` instead of `update_table(State,X,Table)`), so that helpers can be more easily discriminated from member methods, which have such a state as first parameter
- all class-specific attributes shall be documented in a proper `class_attributes` define, so that their name, type, meaning and role are described

[2] It can be generated by running `make full-doc` from `myriad/doc`.

Example:

```
% The class-specific attributes of an instance of this class are:
-define( class_attributes, [

   { current_step, step_count(),
     "current step at which the experiment is" },

   [...]

]).
```

1.7 Traces Conventions

So that log messages can be kept track of over time, a distributed trace system is provided, with relevant components: trace emitter, listener, supervisor, aggregator, etc.

At implementation time, one just has to choose:

- the trace channel on which the trace should be sent, among: `fatal`, `error`, `warning`, `info`, `trace`, `debug` (from highest priority to lowest)
- if a constant message is to be sent (ex: `?warning("This is a static message")`), or if it is determined at runtime (ex: `?warning_fmt("There are ~B apples.",[Count])`)
- if the trace is sent from a method (ex: `?info("Hello")`) or from a constructor (ex: `?send_info(MyState,"Hello")` where `MyState` is a state returned, directly or not, by a `class_TraceEmitter` constructor)
- there are other, less commonly used, information that can be specified, as the categorisation (`_cat` variations, like in `?debug_cat("Hello","core.greetings")`), additional timing information (`_full` variations, like in `?trace_full("Bye","core.greetings",_Tick`) etc.

Once building the simulator, one can configure:

- whether traces should be disabled or enabled (the default), by commenting-out appropriately `-define(TracingActivated,).` in `class_TraceEmitter.hrl`
- if enabled, what kind of trace output will be generated, among LogMX-compliant (a third-party log supervisor integrating a trace parser of our own), PDF output, or raw text output; this is to be set in `traces.hrl` (default is: LogMX-compliant)

At execution time, the command-line option `--batch` can be specified, which causes all interactive elements to be disabled, including any trace supervisor (like the LogMX browser). It can be specified that way:

```
$ make my_case_run CMD_LINE_OPT="--batch"
```

For convenience, often the developer defines in his shell environment file `export BATCH='CMD_LINE_OPT="--batch"'`, so that he can make use of this shorthand instead:

```
$ make my_case_run $BATCH
```

(this is very convenient when debugging brand new code: before having to peer at the traces, runtime errors may occur, and the relevant information, like the stack trace, the actual parameters and the current instance state will be displayed on the console)

The trace system has been designed with performance and scalability in mind, thus if disabled no per-instance overhead penalty will be incurred at all, and, if enabled, efforts have been made so that as large as possible a number of traces are to be managed by the trace system, for any given resources in terms of network, memory and processing.

Should numerous traces be sent, it could be interesting to create more than one instance of the trace aggregator class, for example:

- one for the technical traces, to ease lower-level debugging
- one for the model-centric traces, to ease the debugging of the behaviours and interactions of actors

An overloaded trace aggregator will notify the user, so that the verbosity of later runs can be decreased.

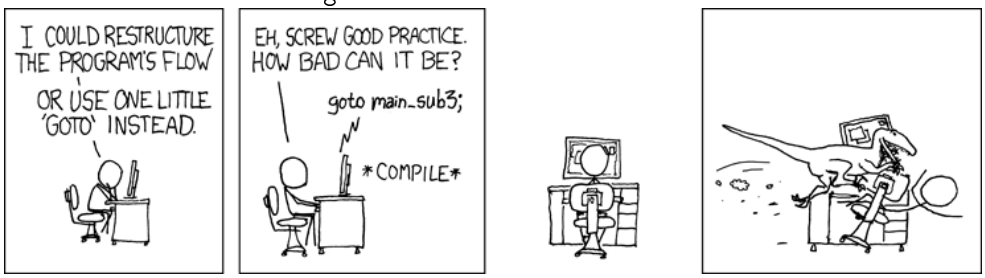
1.8 Sim-Diasca Conventions

In this section the conventions to be respected with regard to Sim-Diasca are detailed.

1.8.1 Thou Shalt Not Bypass The Simulation Engine

The Sim-Diasca services should be used whenever applicable. Notably, bypassing the simulation mechanisms (for actor creation, communication, deletion, etc.) is absolutely prohibited, as essential properties, like the respect of causality, would then be lost.

Such a short-time "simplification" would be considered still more harmful than others like the use of the `goto` statement in `C`:



1.8.1.1 Proper Inter-Actor Communication

To implement the communication between two actors, neither plain Erlang messages nor arbitrary WOOPER messages can be used: as we aim to develop a distributed *simulation*, as opposed to a mere distributed *application*, we have to make use of the Sim-Diasca mechanisms for automatic inter-actor message reordering.

This means that, for inter-actor communication purpose, *only* the following helper functions shall be used:

- for the vast majority of cases: `class_Actor:send_actor_message/3`
- for the very special case where the same message is to be sent to a *very large* number of other actors (typically dozens of thousands), from an instance of `class_BroadcastingActor`: one should use `send_actor_message/3`, `send_actor_messages/3` or `send_actor_messages_over_diascas/3`, as defined in the latter class

Of course one may rely on higher-level specialization layers, making use of Sim-Diasca but providing another API. The point is that inter-actor messages should be ultimately managed by Sim-Diasca, rather than being sent directly with the `!` operator provided by Erlang.

1.8.1.2 Proper Actor Life-Cycle

Very similarly, life-cycle of actors should be ultimately managed by the engine, not directly by the developer: the creation and termination of actors must respect the conventions detailed below.

1.8.1.2.1 Proper Actor Creation

An actor can be created either initially (before the simulation is started) or at simulation-time (i.e. in the course of the simulation), as detailed in next sections.

Calling directly (i.e. from the user code) any `spawn` variation or any WOOPER `new` variation (ex: `remote_new_link`) is totally prohibited: we must rely on the Sim-Diasca mechanisms, and not attempt to bypass them.

The actual creation will be performed by the load balancer, on a computing node of its choice, and the placement will be fully transparent for the model writer.

Whether an actor is initial or not, construction parameters are to be supplied for its creation.

These construction parameters will be specified as a list, for example `[_Age=5,_Gender=male]`. The matching constructor *must* then be in the form of:

```
construct( State, ActorSettings, P1, P2 ) ->
```

Note that two additional initial parameters appear here:

- `State` (whose type is `wooper:state()`), which corresponds to the engine-supplied blank initial state this instance starts with
- `ActorSettings` (whose type is `class_Actor:actor_settings()`), which will be provided automatically by the load balancer too, at runtime; this parameter is just to be listed by the model developer in the constructor of the model, when calling its parent constructor which inherits, directly or not, from `class_Actor`, as shown below.

So a typical constructor for a model `class_M1`, inheriting from, for example, `class_SpecialisedActor` and from classes `class_C1` and `class_C2`, could be:

```
% Constructs a new instance of class_M1:
construct( State, ActorSettings, P1, P2, ... ) ->
  % Will result ultimately in a call to
  % class_Actor:construct( State, ActorSettings, AName ):
  SpecialisedState = class_SpecialisedActor:construct( State,
   ActorSettings, ... ),
  C1State = class_C1:construct( SpecialisedState, P1, ... ),
  C2State = class_C2:construct( C1State, P1, P2, ... ),
  [...]
  FinalState.
```

Note how states are chained (one being built on top of the other), from the blank, initial one (`State`) to the one corresponding to the complete initial state of the instance (`FinalState`), as returned by the constructor.

We strongly encourage the use of type specifications, which would be here:

```
-spec construct( wooper:state(), class_Actor:actor_settings(),
     type_of_p1(), type_of_p2() ) -> wooper:state().
```

1.8.1.2.1.1 Initial Actor Creation

1.8.1.2.1.1.1 Basics

An abstraction API is available to create from a simulation case *initial* actors, i.e. bootstrapping actors, which are created before the simulation is started.

It is generally based on the `class_Actor:create_initial_actor/2` static method:

```
ActorPid = class_Actor:create_initial_actor( ActorClassName,
   ActorConstructionParameters )
```

For example, in `my_example_test.erl` we could have:

```
ActorPid = class_Actor:create_initial_actor( class_PinkFlamingo,
  [ _Age=5, _Gender=male ] )
```

Should multiple initial actors have to be created, using this method would be less than optimal, as the load-balancer would be looked-up in the process registry at each call of this static method, which, if creating thousands of actors in a row, could induce some overhead.

Therefore a more efficient alternative is available, the `class_Actor:create_initial_actor/3` static method, for which the PID of the load-balancer is to be specified as a parameter, having thus to be looked-up only once in the simulation case:

```
LoadBalancerPid = LoadBalancer:get_balancer(),
FirstActorPid =  class_Actor:create_initial_actor( Class1, Parameters1,
  LoadBalancerPid),
SecondActorPid = class_Actor:create_initial_actor( Class2, Parameters2,
  LoadBalancerPid),
[...]
```

1.8.1.2.1.1.2 Multiple Parallel Creations

A typical use case is to load from any source (file, database, etc.) a set of construction parameters for a large number of instances.

For larger cases, creating actors sequentially may lead to very significant simulation start-up durations.

In such cases, `class_Actor:create_initial_actor/1` should be used instead : then a smart, parallel, batched creation will be done, allowing to create all instances as efficiently as reasonably possible.

This results in a considerably faster creation of the initial state of the simulation, provided there is no dependency between the created actors in the specified batch. Otherwise actors should be created in multiple stages, to ensure that the PID of the prerequisite actors is already known and can be specified at a later stage, when in turn creating the actors whose constructor requires these PIDs.

For non-programmatic, file-based initialisation, we strongly recommend using our rather advanced loading system, as described in the technical guide (see its `Sim-Diasca Management of Simulation Inputs` section).

1.8.1.2.1.1.3 Synchronicity

All initial operations (i.e. all operations to be triggered before the simulation starts) must be synchronous, to ensure they are indeed finished once the simulation is run: the simulation case has to wait for their completion before greenlighting the start the simulation.

This involves the use of:

- synchronous creations, which is already enforced by the aforementioned `class_Actor:create_initial_actor{2,3}`, etc. static methods
- requests rather than oneways, once instances are created and the simulation case intends to act upon them (for example in order to link them together); requests must be used, not necessarily in order to retrieve a potential result, but at least to ensure that they are fully processed before the simulation starts (hence the need of using a receive; from the simulation case, one shall prefer using `test_receive/0` or `app_receive/0` - both exported by the `Traces` layer - rather than classical `receive` constructs, see below)

Otherwise there could be a race condition between the end of these initial operations (which may take any time) and the triggering of the simulation start (a message which, without flow control, could be sent too early by the simulation case).

1.8.1.2.1.1.4 Nested Creations

When creating initial actors, we might find useful to create an actor A that would create in turn other initial actors, and so on (nested creations).

This is possible, however these creations should not be directly done from the constructor of A, as this would lead to a systematic deadlock by design[3]. Some solutions have been identified, but they were not satisfactory enough[4].

Instead, the constructor of A should just create A and return, and the actual creations of other actors should be triggered by a subsequent method call (a request, not a oneway, as explained in the Synchronicity section).

For example, in `my_creation_test.erl`, we could have:

```
[...]
ActorAPid = class_Actor:create_initial_actor( ClassA,
   ParametersForA ),
ActorAPid ! { createDependingActors, [], self() },
actors_created = test_receive(),
[...]
```

Note that `test_receive/0` corresponds to a safer form than `receive {wooper_result, R} -> R end`. It is logically equivalent, but immune to interfering messages that could be sent to the simulation case by other Sim-Diasca services (ex: notifications from the trace supervisor).

1.8.1.2.1.2 Simulation-time Actor Creation

[3] A deadlock will occur because the load balancer will be blocked waiting for the creation of actor A to finish, thus paying no attention to the requested creations in-between, while they themselves are waited for the creation of A to complete.

[4] A non-blocking solution could be to have a load balancer that does not wait for an instance to acknowledge that its spawn is over: the load balancer would thus return immediately and keep track of the `spawn_successful` message (interpreted as a oneway) that it should receive before the simulation starts.

However in that case no total order in actor creation seems to be possibly guaranteed: actor A could create B and C, which themselves could, after some processing, create others actors. As a consequence B and C would create them concurrently, and, depending on various contextual factors, their creation requests could be received by the load balancer in no particular order, leading to a given actor bearing different AAI from one run to another. Nested creations would thus be obtained at the expense of reproducibility, which is not wanted.

Once the simulation is started, an actor can *only* be created by another one (for example it then cannot be created directly by the simulation case itself), so that a correct simulation time can be enforced.

The creating actor should call the `class_Actor:create_actor/3` helper function for that creation, like in:

```
CreationState = class_Actor:create_actor( Classname,
   ConstructionParameters, State ),
[...]
```

If called at simulation timestamp `{T,D}`, then the specified actor will be actually created (by the load-balancer) at `{T,D+1}`, and at `{T,D+2}` the creating actor will know (as its `onActorCreated/5` method will be called) the PID of the just created actor.

The creating actor - and any other actor that will be given the returned PID - can then freely interact with the created actor (of course thanks to actor messages), exactly as with any other actor (once its creation is performed, there is no difference between an actor created in the course of the simulation and an initial actor).

1.8.1.2.1.3 Creation With Placement Hints

Regardless of whether a creation is to happen initially or on the course of the simulation, it is often a lot more efficient to ensure that sets of actors known to be tightly coupled are created on the same computing host (i.e. are co-allocated).

Otherwise these actors would be scattered by the load balancer on multiple computing hosts according to its placement policy, i.e. regardless of their relationship (since the load balancer has no a priori knowledge about the interactions between models), which would lead in the general case to a useless massive network overhead, and thus to simulations that would be considerably slowed down.

Sim-Diasca offers a way of forcing co-allocation (i.e. to ensure that a set of actors will be in all cases created on the same computing host, no matter of which host it is), thanks to *placement hints*.

A placement hint can be any Erlang term (atoms are generally used for that purpose), that can be specified whenever an actor is created. The engine guarantees that two actors created with the same placement hint will end up being instantiated (by the load balancer) on the same computing host[5].

So Sim-Diasca provides a counterpart to its basic creation API, whose functions are just expecting one extra parameter, the aforementioned placement hint:

- `class_Actor:create_initial_actor/{2,3}` have `class_Actor:create_initial_placed_actor/{3,` counterparts
- `class_Actor:create_actor/3` has a `class_Actor:create_placed_actor/4` counterpart

Except the hint specification, these functions work exactly as their counterpart (ex: w.r.t. the call to `onActorCreated/5`).

[5] Unless a compute node was lost in the course of a simulation that recovered from it.

For example, if devices in a house were to be modelled, and if a large number of houses was to be simulated, then for house 437, the placement hint (as an atom) `house_437` could be specified for the house creation, as well for the creation of each of the devices it will contain.

That way they would be all created and evaluated on the same computing host, exchanging numerous local messages with no need for costly and slow networked messages.

1.8.1.2.2 Proper Actor Termination Removing an actor from the simulation is a bit more complex than inserting a new one, due to pending inter-actor relationships that may interfere with the actor termination.

An actor A should not decide that another actor B is to be removed immediately from the simulation. Notably, sending a `delete` message to B means just calling directly the WOOPER destructor and therefore bypassing the Sim-Diasca simulation layer and making the simulation freeze or fail on error[6].

Instead the actor A should send an actor message to actor B (if ever B is not just to terminate solely on its own purpose), resulting on the corresponding oneway of B to be triggered. Then B may or may not choose to terminate, immediately or not. Alternatively B may, by itself, determine it is time for it to be removed from the simulation.

In any case, B will decide that it terminates, at `{T,D}`. The main conditions for its deletion is that:

- there is no more spontaneous action that is planned for it: actor B should not plan anymore a future action, and it should withdraw from its time manager any already-planned future action(s); on termination this will be checked by the time manager, which would then trigger a fatal error if at least one spontaneous action was found for the terminating actor

- no other actor will ever try to interact with it (i.e. with B) once it will have terminated; for that, usually B has to notify other actors of its termination, so that they can "forget" it (to ensure that they will never attempt to interact with B again); it is up to the corresponding models to ensure of such an agreement, based on the deferred termination allowed by the API detailed below

To emphasize more, the model developer should ensure that, once an actor is terminated, no other actor expects to interact with it anymore (i.e. that all other actors should stop sending actor messages to it). The objective is therefore to delay appropriately the triggering of the termination of an actor until all possibilities of outside interactions are extinguished.

The smallest duration for a termination procedure cannot be automatically determined, as the PID of the terminating actor (B) can have been transmitted in the meantime from actors to actors. Therefore it is the duty of the developer to ensure that a terminating actor B is safely unregistered from all the actors that may interact with it in the future (generally a small subset of the ones that know its PID). Often this unregistering procedure is best done directly from the actor B itself. Then only B can safely terminate.

Two options exist for a proper termination procedure:

- either to simply postpone the deletion of B until the end of the current *tick* (`T`), letting all diascas that are needed in-between elapse, so that the aforementioned forgetting can take place

[6] Indeed actor B would then terminate immediately, either causing the time manager to wait for it unsuccessfully (if the tick of B was not finished yet) or possibly making it be removed from the simulation whereas another actor could still send an actor message to it, thus being blocked forever, waiting for an acknowledgment that would never come. Moreover the time manager intercepts actor deletions and checks that they were indeed expected.

- or to finely tune the waiting over diascas so that *B is deleted as soon as strictly needed* (i.e. as soon as all potential actors aware of B know now that B is terminating), before even the end of the current tick; in this case the number of diascas to wait depends on the length of the chain of actors knowing B (i.e. actor C may know B and may have transmitted this knowledge to D, etc.)

The first option is by far the simplest and most common: B simply calls `class_Actor:declareTermination/1`, and, starting from the same diasca, notifies any actor of its deletion. The notification chain will unfold on as many diascas as needed, and once all the diascas for the current tick will be over, a new tick will be scheduled and B will then be deleted automatically.

The second option is more precise but more demanding, as it requires B to be able to determine an upper-bound to the number of diascas that can elapse before it can safely terminate (thus without waiting for the next tick to happen).

Such a feature is provided so that, during a tick, any number of actor creations, deletions and interactions may happen, "instantaneously", and according to any complex pattern.

For example, B may know that only actor C knows it, in which case B will notify C of its termination immediately, implying that starting from `{T,D+2}` C is expected to never interact with B anymore (C will receive and process the message at `{T,D+1}` but due to message reordering C might already have sent a message to B at this timestamp - in the general case B should ignore it).

In this context B is to call `class_Actor:declareTermination/2`, with a termination delay of 2 diascas. A larger delay would have to be specified if C had to notify in turn D, and so on...

With both termination options, once `class_Actor:declareTermination/{1,2}` is called, the engine will take care of the appropriate waiting and then of the corresponding deletion, with no further intervention.

Note that:

- should a too short termination delay be chosen by mistake, the simulation engine will do its best to detect it

- if setting up a proper termination happens to be too cumbersome on to many cases, an automatic system might be designed, in order to keep track of inter-model references (ex: like a garbage collector operated on actors, based on reference counting - either PID or AAI); however this mechanism would probably have some major drawbacks by design (complex, expensive because of reference indirections, etc.); moreover having an implicit, dynamic, flexible communication graph is probably more a feature than a limitation

Note

The proper termination of an actor results into a *normal* termination, not in a crash. Therefore processes (including other actors) that would be linked to a terminating actor will *not* be terminated in turn because of it.
On the other hand, as soon as an actor crashes, the simulation is expected to fully crash in turn, in order to avoid silent errors; knowing that anyway no automatic fall-back to a crash can be defined, since it generally means there is a bug in the code of at least a model.

1.8.1.2.3 Summary of The Sim-Diasca Conventions to Enforce

1.8.1.2.3.1 Regarding State

In the code of an actor (i.e. inheriting from a `class_Actor` child instance), the only attributes inherited from Sim-Diasca that should be directly accessed from models is `trace_categorization`, to provide from the constructor various ways of selecting trace messages afterwards.

All other attributes inherited from a `class_Actor` instance should be regarded as strictly private, i.e. as technical details of the engine that are not of interest for the model developer (neither in terms of reading nor of writing).

Of course the developer is free of defining any class hierarchy, with each specialising class defining all (non-colliding) attributes needed.

1.8.1.2.3.2 Regarding Behaviour

Action	**Correct**	**Incorrect**
Initial Actor Creation (before the simulation start)	`class_Actor:create_initial_actor/2` (directly from the simulation case)	Use of a variation of `spawn` or of WOOPER `new`
Runtime Actor Creation (in the course of the simulation)	`class_Actor:create_actor/3` (only from another actor)	Use of a variation of `spawn` or of WOOPER `new`
Actor Communication	`class_Actor:send_actor_message/3`	`TargetActor ! AMessage`
Actor Termination Decision	Notify relevant actors and postpone termination until longest possible interaction is necessarily over	Immediate non-coordinated triggered termination
Actor Termination Execution	`class_Actor:declareTermination/[1,2]`	`TargetActor ! delete`

1.8.2 Actor Scheduling

1.8.2.1 Basics

1.8.2.1.1 Simulation Time: Of Ticks and Diascas Simulation time is fully decorrelated from wall-clock time, and is controlled by the time manager(s): the `fundamental frequency` of the simulation (ex: 50Hz) leads to a unit time-step (a.k.a. `simulation tick`) to be defined (ex: 20ms, in simulation time), each time-step lasting, in wall-clock time, for any duration needed so that all relevant actors can be evaluated for that tick.

If that wall-clock duration is smaller than the time-step (the simulation is "faster than the clock"), then the simulation can be `interactive` (i.e. it can be slowed down on purpose to stay on par with wall-clock time, allowing for example for some human interaction), otherwise it will be in `batch` mode (running as fast as possible).

A simulation tick is split into any number of logical moments, named `diascas`, which are used to solve causality and are not associated to any specific duration by themselves.

Both ticks and diascas are positive unbounded integers.

So a typical simulation timestamp is a tick/diasca pair, typically noted as `{T,D}`.

1.8.2.1.2 Time Managers Controlling this simulation time means offering a scheduling service, here in a distributed way: it relies on a tree of time managers, each being in charge of a set of direct child managers and of local actors.

This scheduling service drives them time-wise, so that they all share the same notion of time (ticks and diascas alike), find a consensus on its flow, while still being able to evaluate all corresponding actors in parallel, in spite of their possible coupling.

In the most general terms, the behaviour of an actor is partly determined by what it would do by itself (its "spontaneous behaviour"), partly by the signals its environment sends to it, i.e. based on the messages that this actors receives from other actors (its "triggered behaviour").

In both cases, for an actor, developing its behaviour boils down to updating its state and/or sending messages to other actors, and possibly planning future spontaneous actions and/or sending information to probe(s).

1.8.2.1.3 At Actor Creation Each actor, when created, has first its `onFirstDiasca/2` actor oneway triggered[7]. This is the opportunity for this newly created actor to develop any immediate first behaviour, and also to specify at once when it is to be scheduled next for a spontaneous behaviour: otherwise, as all actor are created with an empty agenda, they would remain fully passive (never being spontaneously scheduled), at least until a first actor message (if any) is sent to them.

So all models are expected to define their `onFirstDiasca/2` actor oneway[8], in which most of them will at least program their next spontaneous schedulings

[7] This actor actually receives the corresponding actor message sent by the load balancer, which determined a placement for it and created it.

(see, in `class_Actor`, notably `addSpontaneousTick/2` and `addSpontaneousTicks/2`). This corresponds, internally, to exchanges with the time managers in charge of the corresponding actors.

Creations happen at the diasca level rather than at the tick level, so that any sequence of model-related operations (creation, deletion, action and interactions) can happen immediately (in virtual time), to avoid any time bias.

1.8.2.1.4 Afterwards Then a very basic procedure will rule the life of each actor:

1. when a new simulation tick `T` is scheduled, this tick starts at diasca `D=0`
2. as the tick was to be scheduled, there was at least one actor which had planned to develop a spontaneous behaviour at this tick; all such actors have their `actSpontaneous/1` oneway executed
3. as soon as at least one actor sent an actor message, the next diasca, `D+1`, is scheduled[9]
4. all actors targeted by such a message (sent at `D`) process their messages at `D+1`; possibly they may send in turn other messages
5. increasing diascas will be created, as long as new actor messages are exchanged
6. once no more actor message is sent, the tick `T` is over, and the next is scheduled (possibly `T+1`, or any later tick, depending on the spontaneous ticks planned previously)
7. simulation ends either when no spontaneous tick is planned anymore or when a termination criteria is met (often, a timestamp in virtual time having been reached)

Internally, these scheduling procedures are driven by message exchanges by actors and time managers:

- when a tick begins (diasca zero), each time manager sends a corresponding message to each of its actors which are to be scheduled for their spontaneous behaviour
- when a (non-zero) diasca begins, actors that received on the previous diasca at least one actor message are triggered by their time manager, so that each actor can first reorder appropriately its pending messages on compliance with the expected simulation properties (notably: causality, reproducibility, ergodicity), and then process them in turn

[8] Knowing that the default implementation for `onFirstDiasca/2`, inherited from `class_Actor`, simply halts the simulation on error, purposely.

[9] Actually there are other reasons for a diasca to be created, like the termination of an actor, but they are transparent for the model developer.

1.8.2.2 Actor Scheduling

The basic granularity in virtual time is the tick, further split on as many diascas as needed (logical moments).

The engine is able to automatically:

- jump over as many ticks as needed: ticks determined to be idle, i.e. in which no actor message is to be processed, are safely skipped
- trigger only the appropriate actors once a diasca is scheduled, i.e. either the ones which planned a spontaneous behaviour or the ones having received an actor message during the last diasca or being terminating
- create as many diascas during a tick as strictly needed, i.e. exactly as long as actor messages are exchanged or actors are still terminating

Indeed the simulation engine keeps track both of the sendings of actor messages[10] and of the planned future actions for each actor. It can thus determine, once a diasca is over, if all next diascas or even a number of ticks can be safely skipped, and then simply schedule the first next timestamp to come.

So, for any simulation tick, each actor may or may not be scheduled, and an actor will be scheduled iff:

- it planned a spontaneous behaviour for this diasca
- or it received at least one actor message during the last diasca
- or it is terminating

The actor happens to be itself able to keep track of its expected schedulings, and thus it can automatically check that they indeed match exactly the ones driven by the time manager, for an increased safety.

Anyway these mechanisms are transparent to the model developer, who just has to know that all actor messages, once appropriately reordered, will be triggered on their target, and that the planned spontaneous schedulings will be enforced by the engine, according to the requests of each actor.

Thus the developer just has to define the various actor oneways that the model should support (i.e. the ones that other actors could trigger thanks to an actor message), and the spontaneous behaviour of that model (i.e. its `actSpontaneous/1` oneway). Then the simulation engine takes care of the rest.

1.8.2.3 Planning Future Spontaneous Behaviour

Each actor is able to specify, while being scheduled for any reason (an actor message having been received, and/or a spontaneous action taking place), at least one additional tick at which it should be spontaneously scheduled later. An actor can be scheduled for a spontaneous action up to once per tick.

To do so, it can rely on a very simple API, defined in `class_Actor`:

[10] This is done on a fully distributed way (i.e. through the scheduling tree of time managers over computing nodes) and all communications between an actor and its time manager are purely local (i.e. they are by design on the same Erlang node).

Moreover the messages themselves only go from the emitting actor to the recipient one: in each diasca, only the *fact* that the target actor received a first message is of interest, and this is reported only to its own, local time manager - the actual message is never sent to third parties (like a time manager), and no more notifications are sent by the receiving actor once the first message has been reported. So the number of messages, their payload and communication distance are reduced to a bare minimum.

- `scheduleNextSpontaneousTick/1`: requests the next tick to be added to the future spontaneous ticks of this actor
- `addSpontaneousTick/2`: adds the specified spontaneous tick offset to the already registered ones
- `addSpontaneousTicks/2`: same as before, this time for a *list* of tick offsets
- `withdrawSpontaneousTick/2`: withdraws the specified spontaneous tick offset from the already registered ones
- `withdrawSpontaneousTicks/2`: same as before, this time for a *list* of tick offsets

An actor may also decide instead to terminate, using `declareTermination/{1,2}` for that, once having withdrawn any spontaneous ticks that it had already planned[11].

1.8.3 Data Management

In a distributed context, on each computing host, the current working directory of the simulation is set automatically to a temporary root directory, which will be appropriately cleaned-up and re-created.

This root directory is in `/tmp`, to store all live data, deployed for the simulation or produced by it.

Its name starts with `sim-diasca` (to prevent clashes with other applications), then continues with the name of the simulation case (so that multiple cases can run in the same context), then finishes with the user name (so that multiple users can run the same cases on the same hosts with no interference).

Thus the root directory of a simulation on any host is named like:

```
/tmp/sim-diasca-<name of the simulation case>-<user name>
```

For example:

```
/tmp/sim-diasca-Sim-Diasca_Soda_Integration_Test-boudevil
```

This root directory has two sub-directories:

- `deployed-elements`, which corresponds to the content of the simulation package (i.e. both code and data, both for the engine and for the third-party elements, if any)
- `outputs`, which is to contain all live data produced by the simulation (ex: data file, probe reports, etc.); all computing nodes will have directly this directory as working (current) directory

A simulator which added third-party data to the simulation archive (thanks to the `additional_elements_to_deploy` field of the deployment settings specified in the simulation case) is able to access to them thanks to `class_Actor:get_deployed_root_directory/`

For example, if the following was specified:

[11] The time management service could be able to determine by itself which ticks shall be withdrawn whenever an actor departs, however this operation would not be scalable at all (it would become prohibitively expensive as soon as there are many actors and/or many ticks planned for future actions).

```
DeploymentSettings = #deployment_settings{
  ...
  additional_elements_to_deploy = [
    {"mock-simulators/soda-test",code},
    {"mock-simulators/soda-test/src/soda_test.dat",data}
  ...
},
...
```

Then all models are able to access to the data file thanks to:

```
DataPath = file_utils:join( class_Actor:get_deployed_root_directory(State),
  "mock-simulators/soda-test/src/soda_test.dat" ),
% Then open, read, parse, etc. at will.
```

On simulation success, all results will be appropriately generated (in a rather optimal, parallel, distributed way), then aggregated and sent over the network to the centralised result directory, created in the directory from which the simulation was launched, on the user host.

Finally, on simulation shutdown, the deployment base directory will be fully removed.

2 Sim-Diasca Implementation Spotlights

In this section, various technical pieces of information (not of interest for users, but relevant for engine maintainers) will be discussed regarding the *mode of operation* of Sim-Diasca.

2.1 About Erlang Nodes and Simulation Identifiers

2.1.1 How Many Erlang Nodes Are Involved in a Simulation?

By default (unless the case specifies otherwise), only the local host is involved, yet there are two VMs (Erlang virtual machines) running then: the one of the user node, and the one of a (local) computing node.

In the general case, distributed simulations running on `N` hosts will involve by default `N+1` nodes: one user node (on the user host) and `N` computing nodes (including one on the user host).

See the `computing_hosts` field in the `deployment_settings` record (defined in `class_DeploymentManager.hrl`) for further options.

2.1.2 How Are Launched the Erlang nodes?

By default, long names are used for all Sim-Diasca related nodes.

To avoid any possible cross-talk, we have to ensure that a simulation (live or post-mortem) remains fully self-contained and immune to interferences (notably from other simulations that would be run in parallel or afterwards, i.e. both at runtime and regarding the on-disk generated information).

Erlang ensures (thanks to EPMD) that, on any given host, regardless of the Erlang installations, of their version, of the UNIX users involved, no two nodes can bear the same (long) name (otherwise the second node will fail to launch).

The Sim-Diasca **user node** is launched from the generic makefile infrastructure, resulting in the `myriad/src/scripts/launch-erl.sh` helper script to be run with proper parameters.

The name of such a user node (made a distributed node programmatically, see Node Naming) will follow the following format: `Sim-Diasca-<name of the test or case>-<user name>-<simulation instance identifier>-user-node`.

For example, a case named `soda_deterministic_case.erl` run by a user `john` and relying on a Simulation Instance Identifier equal to `43416933` will result in a user node named `Sim-Diasca-soda_deterministic_case-john-43416933-user-node`.

As for the (per-host) **computing nodes**, they will be launched each from their respective `class_ComputingHostManager` instance (driven by the `class_DeploymentManager` singleton created at start-up), and their name will follow that format:

```
Sim-Diasca-<name of the test or case>-<user name>-<simulation instance
   identifier>-computing-node-on-<hostname>
```

The same example, running on host `volt`, will thus result in a computing node to be created under the name:

```
Sim-Diasca-soda_deterministic_case-john-43416933-computing-node-on-volt
```

2.1.3 What is the *Simulation Instance Identifier*?

This information, whose shorthand is `SII`, is a string made of alphanumerical characters, dashes (-) and underscores (_), and is meant to guarantee the uniqueness of a given instance of a simulation (i.e. no two simulations, run simultaneously or not, should ever bear the same SII).

By default the SII is automatically generated and managed by Sim-Diasca. It is based on a UUID (*Universally unique identifier*) determined at start-up[12]. An example of a UUID is `4f8fbacd-93d2-487d-86ff-23f75339c191`.

As the acronym suggests, there is very little chance that two simulation instances may succeed in drawing the same UUID, so they provide an adequate guarantee of uniqueness.

These UUIDs may be deemed a bit too long to be very tractable for humans, so the engine shortens them thanks to hashing (thanks to `erlang:phash2/1`), hopefully preserving a sufficient part of their underlying unicity.

The hash value of our example UUID corresponds to the `43416933` identifier in the previous section.

While such an *automatic identification* is convenient and transparent, for some uses it is possible and even desirable not to rely on such a randomly determined identifier, but to use instead one that is transmitted by a third party (typically if the engine is embedded in a simulation platform able to provide its own identifiers).

Then Sim-Diasca is able to use such an externally-obtained identifier thanks to its `--simulation-instance-id` command line option.

As a result, such a platform may run a simulation case with:

```
$ make my_foobar_case
    CMD_LINE_OPT="--batch --simulation-instance-id 117"
```

Then the specified SII will be used instead of the one that would be determined internally, at runtime, notably to designate:

- the user and computing nodes (ex: `Sim-Diasca-Foobar_Case-john-117-user-node@volt`)
- the simulation result tree (ex: `Foobar_Case-on-2016-6-14-at-17h-07m-28s-by-john-117`)
- the temporary directories (ex: `/tmp/sim-diasca-Foobar_Case-2016-6-14-at-15h-12m-18s-117`)
- the simulation trace file (ex: `Foobar_Case-john-117.traces`)

2.1.4 How Erlang nodes are named?

So the SII is either user-supplied or determined at runtime, by the engine itself. As a result, the name of the user node cannot be determined statically in the general case (the node must run so that it may draw its UUID then determine its SII).

Knowing that additionally a node created as a distributed one (here with the "long names" command-line option) cannot be renamed (`net_kernel:stop/0` not allowed), the only relevant design is, from the Sim-Diasca layer onward

[12] A UUID is obtained thanks to any system-provided `uuidgen` command, otherwise our own implementation is used for that (refer, in the **Myriad** layer, to `basic_utils:generate_uuid/0`).

(lower ones relying on long names) to start the user-node in non-distributed mode (thanks to the `--nn` option of `launch-erl.sh`), establish the name it shall bear, and then only execute `net_kernel:start/1`.

2.1.5 How Is It Ensured that No Two Simulations Can Interfere?

The naming of nodes is a first-level security, which should prevent most accidental collisions.

If ever all other safety measures failed for any reason, a node naming clash will happen, yet it will be detected and will lead to making the clashing simulations fail, so no silent failure shall be feared.

This protection is obtained thanks to Erlang cookies using transparently the UUID mentioned in the previous section (UUIDs will be used in all cases for cookies, even if a third-party SII is specified - for an increased safety, should clashing SIIs be provided by mistake).

3 Sim-Diasca Technical Gotchas

We mention here the main technical sources of puzzlement that may affect the unwary developer.

3.1 The Code Was Updated, Yet Seems To Linger

This may happen **if the code source has been changed yet has not been recompiled before launching a simulation**: Sim-Diasca, once executed, will attempt to compile it, and hopefully succeed.

Then the corresponding BEAM modules will be available in their newer version and, when they will be referenced for the first time, they will be loaded - thus in their newer form.

However, some modules may have already been loaded by the engine (for its internal mode of operation), *before* it triggered the compilation update.

As a result, even if a newer version of their BEAM file becomes available on disk, these modules have already been loaded (and will not be specifically reloaded)[13] ; they will thus stick to their former version, and their newer version will be loaded only at the *next* Sim-Diasca run.

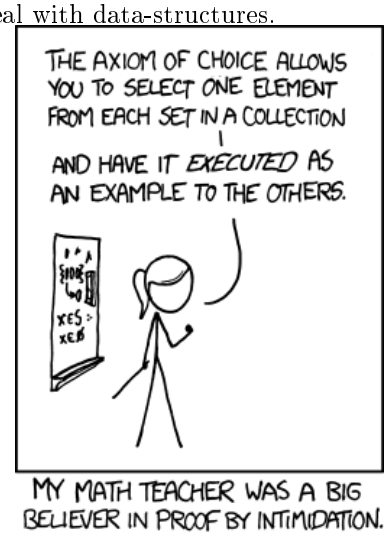
A solution is simply to ensure that any module whose source has been modified is recompiled afterwards (simply a matter of typing `CTRL-P` with our emacs settings), at least *before* a new simulation is run.

[13] Moreover, they may belong to the pioneer modules, in which case they will be deployed over the network on other nodes as well, instead of being read in an updated version from the simulation archive.

4 Developer Hints

4.1 Choosing The Right Datastructures

Writing models involves a lot of algorithmic design decisions, and many of them deal with data-structures.



There is a large choice of both data-structures as such (lists, trees, associative tables, heap, etc.) and of implementations (`gb_tree`, hashtables, etc.), offering various trade-offs.

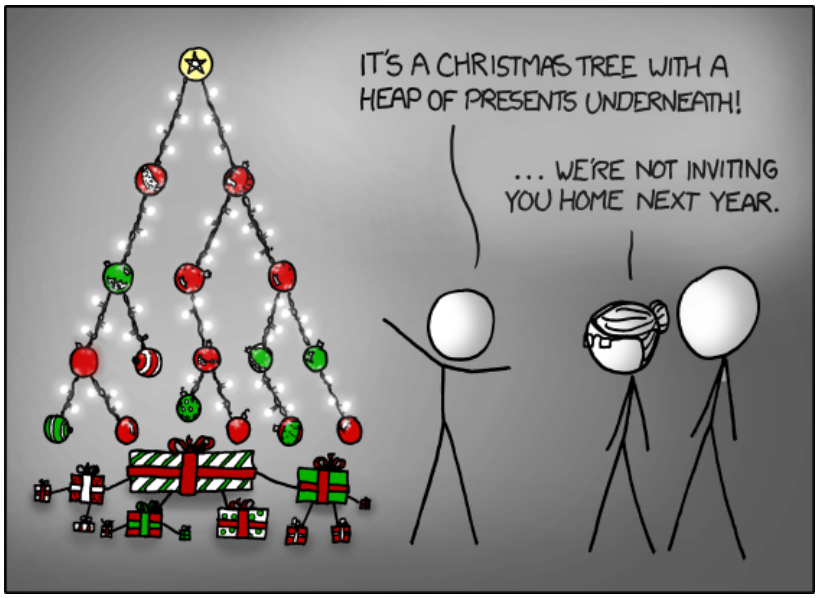
In some occasions, we felt the need to develop our own versions of some of them (see `myriad/src/data-management` for the most common ones), even if in some cases built-in solutions could provide better trade-offs (thinking to ETS tables- albeit offering a different sharing semantics - and to the process dictionary - which is not the purest and most flexible feature we wanted to rely on).

We nevertheless use most of the time the built-in data-structures, like `gb_sets` or `queues`. When multiple implementations providing the same API are available (ex: for ordered lists), we usually define a (sometimes module-specific) `list_impl` symbol, allowing to switch easily between similar datastructures.

For example:

```
% Defines list_impl:
-include("data_types.hrl").

f( A ) ->
   true = ?list_impl:is_empty( A ), [...]
```



Similarly, for **algorithms** operating on these data-structures we tend not to reinvent the wheel (ex: `class_Mesh` uses the `digraph` module), unless we need specific versions of them (ex: operating on an implicit graph, with user-specified anonymous functions, see `myriad/src/utils/graph_utils.erl`).

4.2 Running Bullet-Proof Experiments

Making use of large-scale HPC infrastructures is not straightforward: they often behave like black boxes, and because of the number and specificities of the hardware and software elements that are involved, they tend to exhibit unanticipated behaviours.

Here is a list of recommended steps to go through, in order to have a better chance of making good use of these resources:

1. deactivate the sending of simulation traces, that would otherwise overwhelm the trace aggregator: comment-out, in `traces/src/class_TraceEmitter.hrl`: `%-define(TracingActivated,)`

2. activate any console outputs you are interested in (uncomment relevant `io:format calls`):

   - in `sim-diasca/src/core/src/scheduling/class_LoadBalancer.erl`, in the `create_actor/4` function, if wanting to follow actor creation
   - in `sim-diasca/src/core/src/scheduling/class_DeploymentManager.erl`, in the `launch_node/4` function, if wanting to measure how long it took to accept or reject each candidate host (it may last for more than one full minute in some cases)

3. increase the time-outs:

   - in `wooper/src/wooper.hrl`, one may uncomment the extended `synchronous_time_out` constant recommended for simulation
   - in extreme conditions, in `sim-diasca/src/core/src/scheduling/class_TimeManager.erl`, in the `watchdog_main_loop/3` function, `MaxMinutes` could be increased
   - still in extreme conditions, in `sim-diasca/src/core/src/scheduling/class_DeploymentManage` in the `launch_node/4` function, at the level of the `net_utils:check_node_availability` call, `AttemptCount` could be increased (trade deployment speed for reliability)

4. copy the source of the simulator and update the configuration files according to the case to be run

5. compile everything from the root, from scratch (`make clean all`)

6. for all the classes for which traces are wanted (if any), re-enable their sending:

   - re-activate traces, reverting the content of `traces/src/class_TraceEmitter.hrl`
   - update the time-stamps of all target classes, ex:
     touch sim-diasca/src/core/src/scheduling/class_DeploymentManager.erl

7. re-compile from the root (`make all`) if traces were enabled for at least one class

8. possibly: hide `~/.ssh/known_hosts` to avoid nodes being rejected because of a change in the RSA fingerprint of their key

9. launch in debug mode from the front-end, ex:

```
sim-diasca/conf/clusters/sim-diasca-launcher.sh --debug
 --node-count 32 --cores-per-node 8 --queue parall_256
 --max-duration 64 foobar-simulator/src/uc23_integration_test.erl
```

4.3 Using Type Specifications With Sim-Diasca

4.3.1 Type Specifications: What For?

Adding a type specification (shorthand: *spec*) to the source code of a function means specifying what are its intended input and output parameters, in terms of number and types. This can be applied to records as well.

Once all Sim-Diasca code (including prerequisites, like `Myriad`, `WOOPER` and `Traces`) and the one of user applications based on it (ex: `Mock-Simulators` or any actual simulator) have been instrumented with type specifications (i.e. when all exported functions and records of all modules have a proper spec), then:

- static checkings can be done: Dialyzer is able to detect various discrepancies (such as type errors, unreachable code, unnecessary tests, etc.) at compile-time (therefore a lot earlier than at runtime, and allowing to examine *a priori* all code paths)
- more precise and useful documentation can be generated, thanks to edoc

Some versions of Sim-Diasca are referenced in Dialyzer's application repository.

4.3.2 Type Specifications: How?

4.3.2.1 Prerequisites

4.3.2.1.1 Taking Care of Erlang/OTP First of all, type information must have been already extracted from the Erlang/OTP files of the install that will be used, and stored in a PLT (for *Persistent Lookup Table*) file for later-reuse (this is a preprocessing stage). Such a file is preferably created one time for all for each Erlang environment being used. Therefore a PLT file is better produced as the last step of an Erlang installation (see the `--generate-plt` option of our `install-erlang.sh` script, which streamlines it). This operation is rather long (ex: one hour and a half).

We prefer to have Dialyzer operate on BEAM files (`*.beam`) rather than on source files (`*.hrl/*.erl`), as include paths, symbol definitions and parse transforms are better supported this way.

These BEAM files must have been compiled with debug information (i.e. with the `+debug_info` compiler option). This is thus the default enforced for the full Sim-Diasca software stack.

When writing type specifications, one must know what are the built-in ones, in order to re-use them, so that they do not end up being defined more than once, under different names. To do so, one may use our `myriad/src/scripts/list-available-types.sh` script, like in:

```
$ cd otp_src_RxBy
$ list-available-types.sh | tee declared-types-in-Erlang-RxBy.txt
```

4.3.2.1.2 Taking Care of Above Layers Once a PLT is available for Erlang/OTP, PLTs are to be generated for the entire codebase of interest (typically Sim-Diasca and its prerequisites, and possibly user code as well).

This can be achieved with the `generate-all-plt` make target, to be run from the root of either a check-out or an install. The script will climb our software stack layer by layer, and generate for each a custom PLT (ex: `myriad.plt`).

If, for any reason, the PLT of a layer must be (re)generated, simply use the `generate-local-plt` make target from the root of this layer.

The generation of a PLT will notably allow to catch discrepancies of calls with regard to the specs of the functions being called.

Checking specs against the functions they apply to is useful as well. This can be done on any layer through the `self-check-against-plt` make target.

Like for Erlang, for each layer a repository of the type declarations defined there can be built, either by running `make generate-list-of-local-types` from the root of that layer (producing then a `declared-types-in-<LAYER>.txt` file), or by running `make generate-list-of-all-types` from the Sim-Diasca root to have all lists of types generated at once.

4.3.2.2 Expressing Type Specifications The complete syntax is described here. This will be the main reference to be used and kept ready when writing type specs.

Following conventions are to respect:

- type specifications must be defined in all source files
- all exported functions and records defined in headers should have a type spec, and this spec should be specified on the line just before their own definition; local functions and records may or may not have type specs
- these type specs are to be defined as soon as a new function or record is introduced
- as soon as a data-structure is being used more than once (ex: let's suppose a timestamp is being defined as a triplet of positive integers), a user-specific type *must* be defined (ex: `-type MyTimeStamp() :: {pos_integer(),pos_integer(),pos_i` and re-used *everywhere applicable* (ex: `-spec get_timestamp() -> MyTimeStamp().`)
- all type definitions (opaque or not) must be declared in a relevant module (least astonishment principle), and must be gathered in a section near the top of the file
- types that may be potentially reused elsewhere must be exported; conversely, relevant types that have been already defined must be reused (instead of being defined multiple times); to know what are the currently known types, use our `list-available-types.sh` script
- type specs should include no extraneous whitespaces and should respect the usual 80 character wide lines (thus possibly being broken into multiple lines)
- external data (ex: information input by the user) shall be validated, and the checking code must denote these input information as of type

`basic_utils:external_data()`, `basic_utils:unchecked_data()` or, if appropriate, as a more precise `basic_utils:user_data()`; these types are all (opaque) aliases for `term()`; data shall be tagged with their expected type only once they have been validated (as one should certainly not trust the user or more generally any program interface)

- Dialyzer should be run regularly against the codebase to check frequently whether the sources are correct

For a larger codebase to instrument with type specs, it may be useful to start first with the specs that can be deduced by Dialyzer from the actual code of functions. This can be done thanks to our `add-deduced-type-specs.escript` script (in `myriad/src/scripts`). One should note that these specs are not, in the general case, the ones that the developer would have written (as Dialyzer cannot guess the intent of the original developer), so at least some adaptation work remains (ex: to define reusable types).

> **Note**
> A developer may *overspec* or *underspec*.
> *Overspecification* corresponds to the writing of type specifications that are narrower than the allowed types that a function could process. For example, even if a given function happened to be able to use improper lists as well, the developer may decide that only proper lists are to be passed. Similarly, one may prefer `string()` to `[[any()] | char()]`.
> Reciprocally, *underspecification* corresponds to the writing of type specifications that are larger than the allowed types that a function could process. This may happen if planning to expand later the inputs that a function can take into account.
> Overspecification is perfectly legitimate, whereas underspecification should preferably be avoided.

4.3.2.3 Checking Type Specifications In the context of each layer, one may routinely run:

```
$ make clean all generate-local-plt
```

This allows to list all the types that are unknown (generally misspelled or not exported) and spot a few kinds of errors (ex: `Call to missing or unexported function`).

For a layer `foo` (ex: `Myriad`, `WOOPER`, etc.), one should run from its root directory:

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

You will have an output like:

```
$ make self-check-against-plt
  Building all, in parallel over 8 core(s), from BASE/foo
  [..]
   Checking foo against its PLT (./foo.plt)
 Checking whether the PLT ./foo.plt is up-to-date... yes
```

```
Compiling some key modules to native code... done in 0m29.49s
Proceeding with analysis...
bar.erl:53: Function run/0 has no local return
[..]
```

Issues can then be tackled one by one. To speed up the process of improving a module `bar`, one can run:

```
$ make bar.plt
Checking module 'bar.beam' against relevant PLT
[...]
```

And only this module will be checked, allowing to fix them one by one.

> **Note**
>
> When a source file is modified, the rebuild the BEAM must be triggered specifically, otherwise Dialyzer will not detect that its PLT is not up-to-date anymore (it relies on the timestamp of the BEAM file, not on the one of the `*.erl` file).

4.3.3 References

- Dialyzer homepage
- a useful Dialyzer practical guide
- Types (or lack thereof)
- Types and Function Specifications
- edoc User's Guide

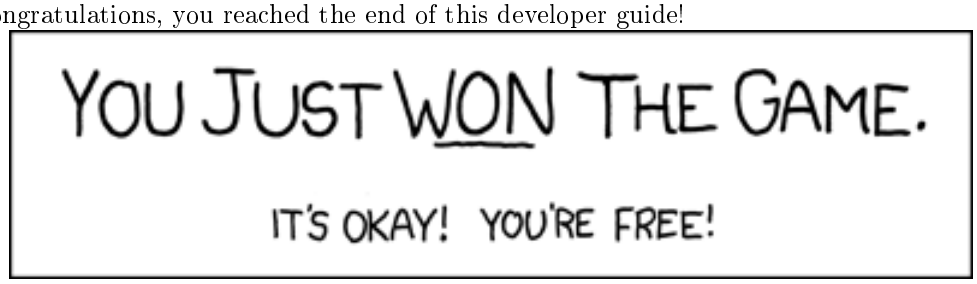
5 Credits

Special thanks to **Randall Munroe** who is the author of all the comic strips that enliven this documentation, and who kindly allowed their use in this material.

See his XKCD website for all information, including for all his delightful other strips.

6 What To Do Next?

Congratulations, you reached the end of this developer guide!



Now you should write your own test models, getting inspiration from the *Sim-Diasca Mock Simulators* (see the top-level `mock-simulators` directory in the source archive).

Writing such toy models is surely the shortest path to the understanding of Sim-Diasca conventions, no matter how fruitless an activity it may seem:

We hope that you will enjoy using Sim-Diasca. As always, any (constructive!) feedback is welcome (use the email address at top of this document for that). Thanks!

Sim-Diasca
Simulation of Discrete Systems of All Scales