Visual Erlang

Torben Hoffmann

Version 0.1.0

Visual Erlang is a notation for describing architectures of Erlang systems [1].

The notation is a dual graphical and textual version with a one to one mapping between the two.

It is not the intention that all details of Erlang programs can be captured in Visual Erlang — the focus is on making it easier to communicate the architecture.

[1] **NOTE:** this is work in progress, so there are things related to both the graphical and textual notation that has to be improved based on the feedback received.

The two building blocks of Erlang — processes and modules – will be used to attach the functionality of a system.

Erlang uses message passing, so there will be a symbol for that in Visual Erlang, but Visual Erlang follows the good practice that most things should be hidden behind an API, e.g., if you have a `gen_server` you will not send messages to it directly using `gen_server:call/2`, but rather provide an API function in a module that will do it for you.

However, there are cases where message passing is the right way to solve a problem, and then it is nice to have a notation for message passing.

The caption of the figures showing how a certain element of Visual Erlang looks like will have the textual version of the same information.

Visual Erlang Symbols

Processes

Visual Erlang uses an ellipse to represent a process since a circle is often hard to size nicely when the process names are long (Figure 1). Luckily, a circle is also an ellipse, so no harm is done if you use a circle.

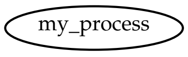

Figure 1: `Process my_process.`

If a process traps exits it should be drawn with a double lined ellipse (Figure2.

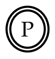

Figure 2: `Process P traps exits.`

Since dealing with failures is a very natural thing in Erlang we have notation for monitoring of processes. When P_1 monitors P_2 it is drawn as a double line ending with a circle (Figure **??**).

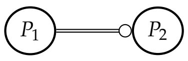

Figure 3: `P1 monitors P2.`

Linking is the dual of monitoring with the extra twist that the linked processes will die together. Linking is drawn with circles at both ends of a double line (Figure 4).

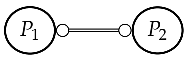

Figure 4: `P1 and P2 are linked.`

Especially for supervision trees it is important which process has spawned another process, so in order to show that P_1 has spawned P_2 an arrow head is drawn towards P_2 (Figure 5).

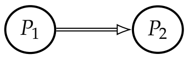

Figure 5: `P1 spawns P2.`

For supervisors it is the case that they both spawns and links to their child processes. This is shown by combining the symbols for spawning and linking (Figure 6).

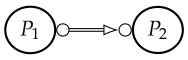

Figure 6: `P1 spawn-links P2.`

Messages and Trap Exit Signals

Showing the sending of messages is meant to be a low-level thing and not to be the guiding principle in Erlang systems. Most of the time message passing will hidden inside modules like the components from the OTP library encourages. In the situations where you have to send a message it is shown as a curly line going from sender to receiver with the message on top of the line (Figure 7).

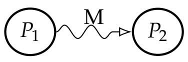

Figure 7: `P1 sends M to P2.`

Trap exit signals are turned into `'DOWN'` messages when a process is trapping exits. The explicit flow of a trap exit message is shown with a zigzag line (Figure 8).

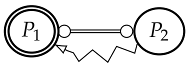

Figure 8: Trap exit handling

This way of showing trap exits does not say much about what happens inside P_1 when the `'DOWN'` message is received. When you want to say something about that you can designate an abstract functionality to a process as shown in Figure 9. “Why not write exactly which part of the code handles the trap exit?” you might ask. The reason is that in many cases you are only interested in seeing what happens as a result of receiving a message, in particular when it is a `'DOWN'` message; exactly which function in a module that handles it is not of interest when communicating at the architecture level.

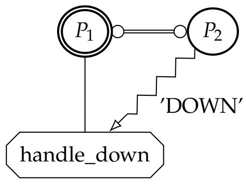

Figure 9: `Process P1 has functionality handle_down. Functionality handle_down handles 'DOWN' message.`

Modules, Processes and APIs

A corner stone in good Erlang programming — or most programming for that matter — is to provide APIs that encapsulates the implementation.

APIs in Erlang are given as exported functions from a module.

A module exporting a function is shown as a double rectangle for the module and a diamond arrow head from the module to the function it exports. Functions are shown as rectangles containing the name and arity of the function. See Figure 10.

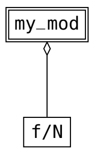

Figure 10: `Module my_mod has API f/N`

Processes are — in general — started through a module. This is shown as a filled diamond arrow head pointing towards the module as in Figure **??**.

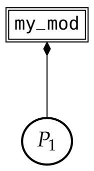

Figure 11: `Module my_mod has instance P1.`

For the majority of the cases the process will be what is referred to and in that case one can use the shorthand in Figure 12, where the name of the module that P_1 is an instance of is given after the process name separated with a colon.

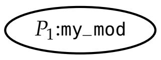

Figure 12: `Process P1 is an instance of my_mod.`

Since the name of the module is often used to refer to the process when there is only one instance of process you can give the name of the module surrounded by colons as a shorthand as in Figure 13.

When an API in a module towards a process always involves interaction with the process it is permitted to associate functions of the module with the process instance as shown in Figure 14.

This is the predominant situation, but in some cases one has functions that do not involve the process and then you see diagrams like in Figure 15. This allows you to clearly communicate which functions of the API that involves the process and which are "merely" regular functions.

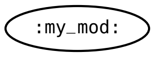

Figure 13: `ModuleProcess my_mod.`

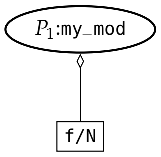

Figure 14: `Process P1 has API f/N.`

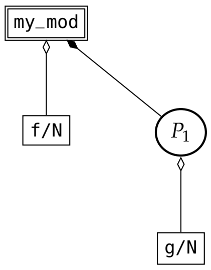

Figure 15: `Module my_mod has instance P1.`
`my_mod has API f/N`
`P1 has API g/N`

Invoking functions

Since APIs are given in terms of functions we need a way to invoke the functions and that is done using an arrow ending with a triangle arrow head from the caller to the function as seen in Figure **??**. Above the arrow you can see the arguments (either the types or some saying names) and below it you can see a short arrow followed by the result to be returned to the caller. If the result isn't important it can be left out.

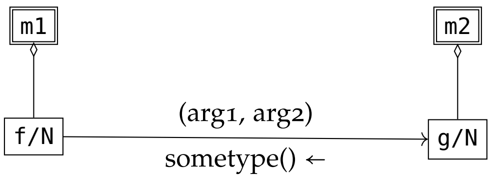

Figure 16: tbd

Missing Notation

- ETS tables.
- states in `gen_fsm`