GenoM3

Anthony Mallet, LAAS - CNRS

Bonn, August 4th, 2010

1. History and concepts
2. Component model
3. Code generation
4. Related tools

Martha EU project – *Circa* 1995

Sara Fleury

Matthieu Herrb

- 1994 - 2000 First version. Running only on VxWorks.
- 2000 - 2004 Linux support with pocolibs.
- 2004 - 2010 Version 2 - Opensource, english documentation. Used by a few other labs/companies (JPL, EPFL, GMV).
- 2010 - Version 3 - Middleware independence.

Salient demonstrations

Hilare2 – 2002

Rackham – 2005

Karma – 2004

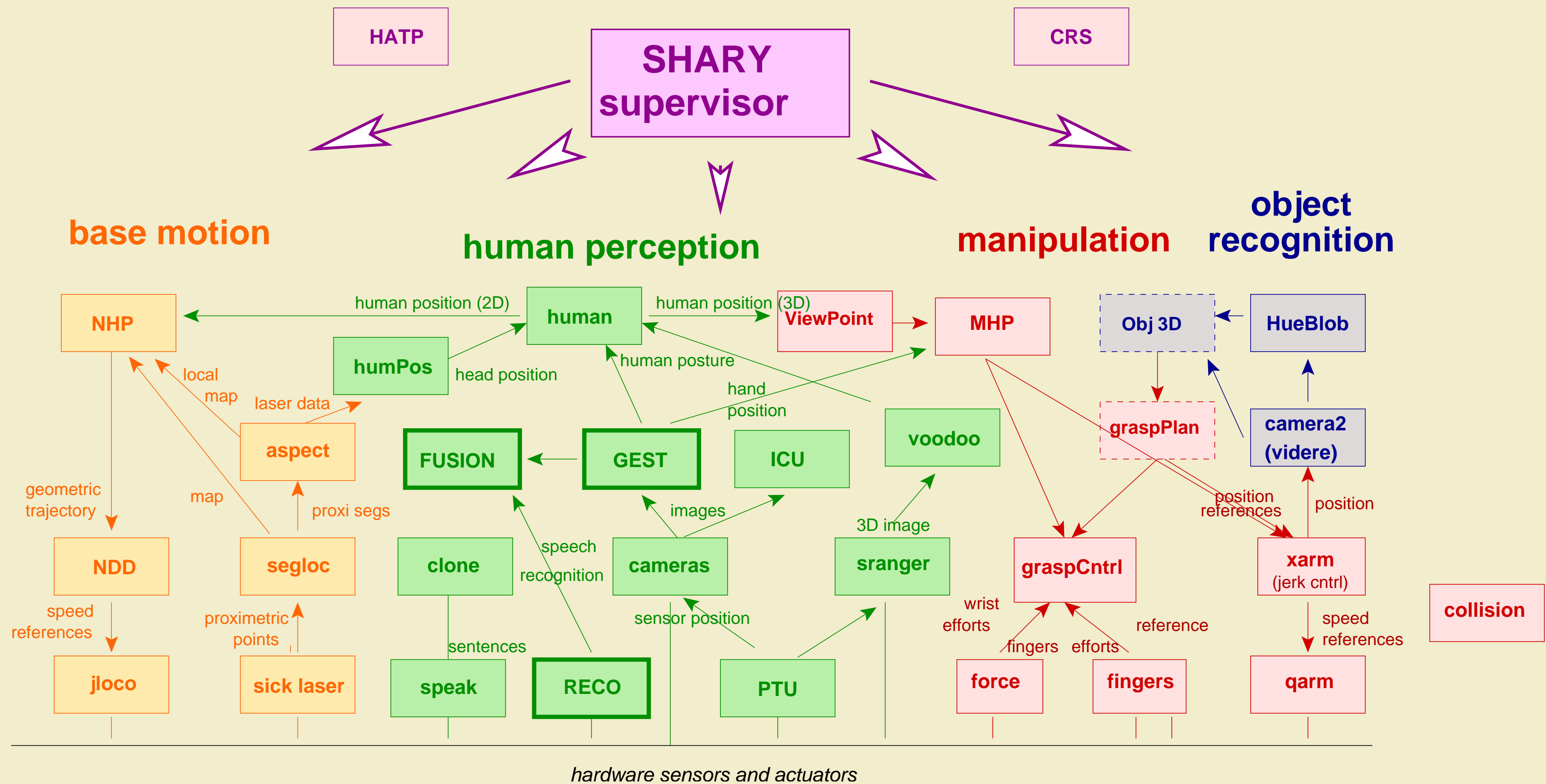
Lama – 2001

Dala – 2004

HRP-2 – 2008

LAAS

LAAS layered architecture: Functions, Supervision



(from the COGNIRON EU project, *circa* 2007)

Components

- are entities relevant to the supervision,
- coarse grained,
- provide high-level services (*e.g.* not a matrix product).

Components formal description

Components are defined by a text file:

- Interface: ports, services.
- Properties.
- Internals:
  - *codels* (elementary code).
  - *executions contexts* (tasks, threads).

$G^{en}oM$: a unique component model

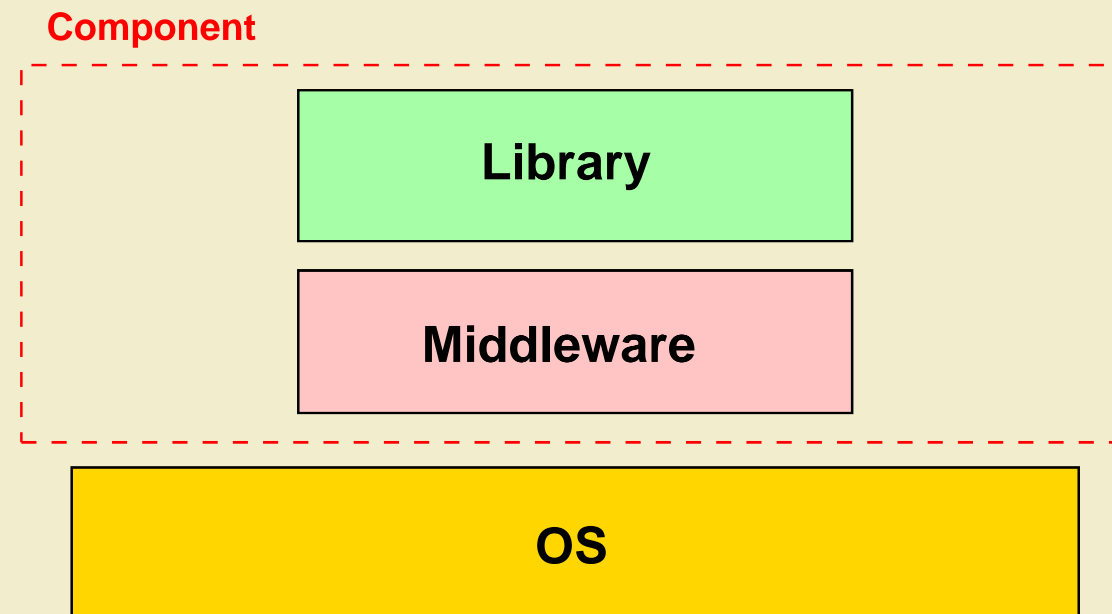
All components share the same component model:

- Robustness
- Ease of development
- Generic supervision architecture
- Software validation
- Decoupling architecture / algorithmic core

- Control flow
  - Service requests, may have parameters and output data.
  - Provides building blocks for application development (supervision).
  - Low bandwidth.
- Data flow
  - Input and output ports: “posters”.
  - Long-lived inter-components connections, configured dynamically via services. Meant for high bandwidth data transmission.

- No control flow between components
  - Prevents conflicting requests.
  - Supervision layer can rely on a known state.
  - Increases modularity.
- Highly restrictive — maybe too much.
  - But never been blocked so far.

A generic component will usually look like this:



There are a lot of different middlewares / frameworks.

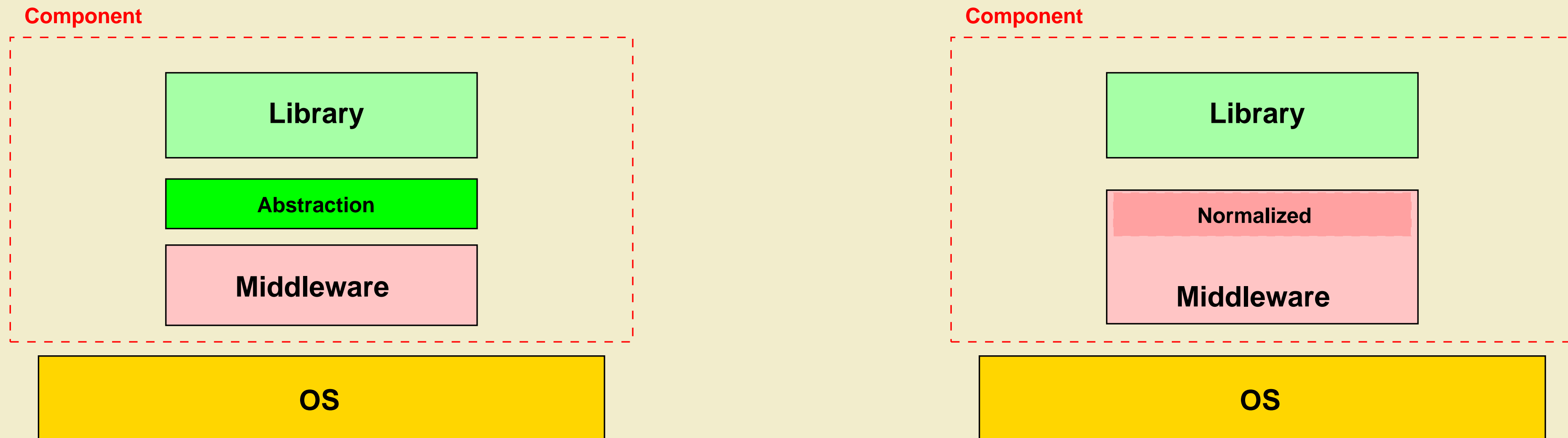
- Player (2001)
- Orocos (2002)
- OpenRTM (2002)
- Urbi (2003)
- Orca (2004)
- YARP (2006)
- ROS (2007)
- ... **ETC** ...

- The choice of a middleware is crucial:
  revoking that choice is costly.
- Different contexts require different solutions.
- There is no unique solution.
- Modularity is good.

Make it easier to use different middlewares ...

... while reusing the same software.

Avoid those approaches:



- An extra layer generates extra code and less performance.
- "Least common denominator" of all the middlewares.
- Helps a lot for communication, but not for the rest.

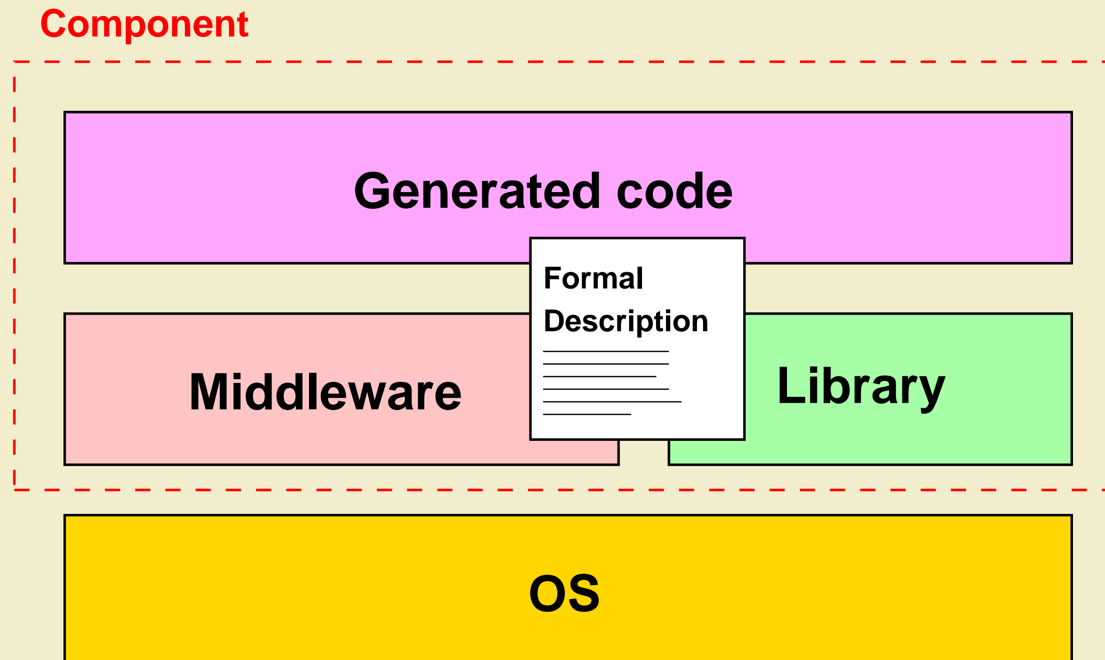
  (component interface definition, internal component behaviour, control architecture definition, ...)

GenoM3 components:

- Can use any middleware.
- Make the underlying library highly reusable.
- Can be compared (e.g. middleware or architecture benchmarks) in the context of real and complex applications.
- Behave according to a common model, allowing generic supervision or validation. e.g. BIP from VERIMAG.

$G^{en}oM3$ component architecture

- No dependency between “library” and “middleware”.
- Middleware-dependent part generated.
- Use of a formal description file.



- Component interface: IDL data structures

```
struct sample_data {
    long l;
    double d;
    string s;
    ...
};
```

- General component description.

```
component name {
    ids:  type_name;
    lang: "c, c++, ...";

    ...
};
```

- Internal Data Structure (IDS)
  - groups all services parameters.
  - state of the component.

- Input/output data ports.

```
inport data<type> name;
ouport data<type> name;

inport event<type> name;
ouport event<> name;
```

- Executions contexts.

```
task name {
   period:         100ms;
   priority:       200;
   stack:          20k;

   codel start:    tstart(in ::ids) yield main;
   codel main:     tmain(in d, out s)
                      yield main, stop;
   codel stop:     tstop() yield ether;
};
```

- Services.

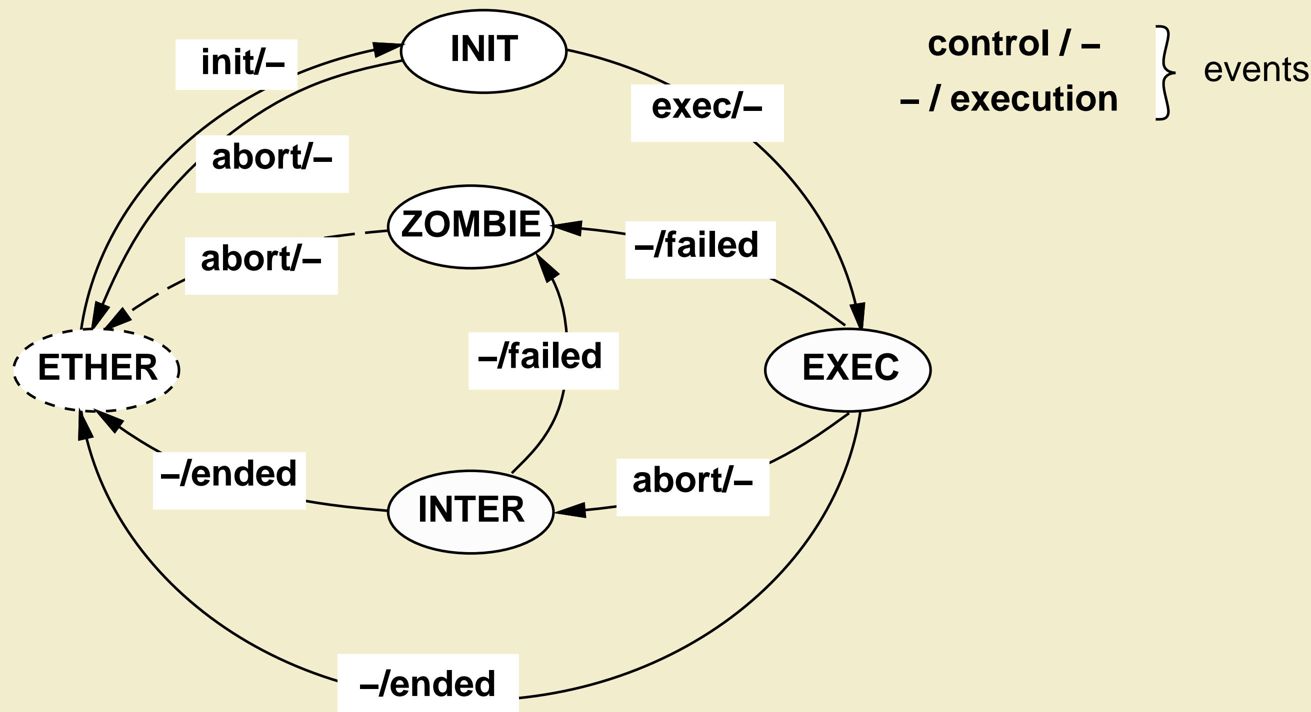
```
service name(inout s) {
   doc:        "Service description";
   task:       name;

   validate:   svalidate();
   throws:     ERROR_1, ERROR_2, ...;
   interrupts: name;

   codel start: sstart() yield step1;
   codel step1: sstep1() yield step1, step2;
   codel step2: sstep2() yield ether;
   codel stop:  sstop() yield ether;
};
```

- A component defines several services.
- Services can be compatible or incompatible.
- Each running service corresponds to an *activity*.
- GenoM3 activities execute according to a state machine.

Example: GenoM2 activity FSM.

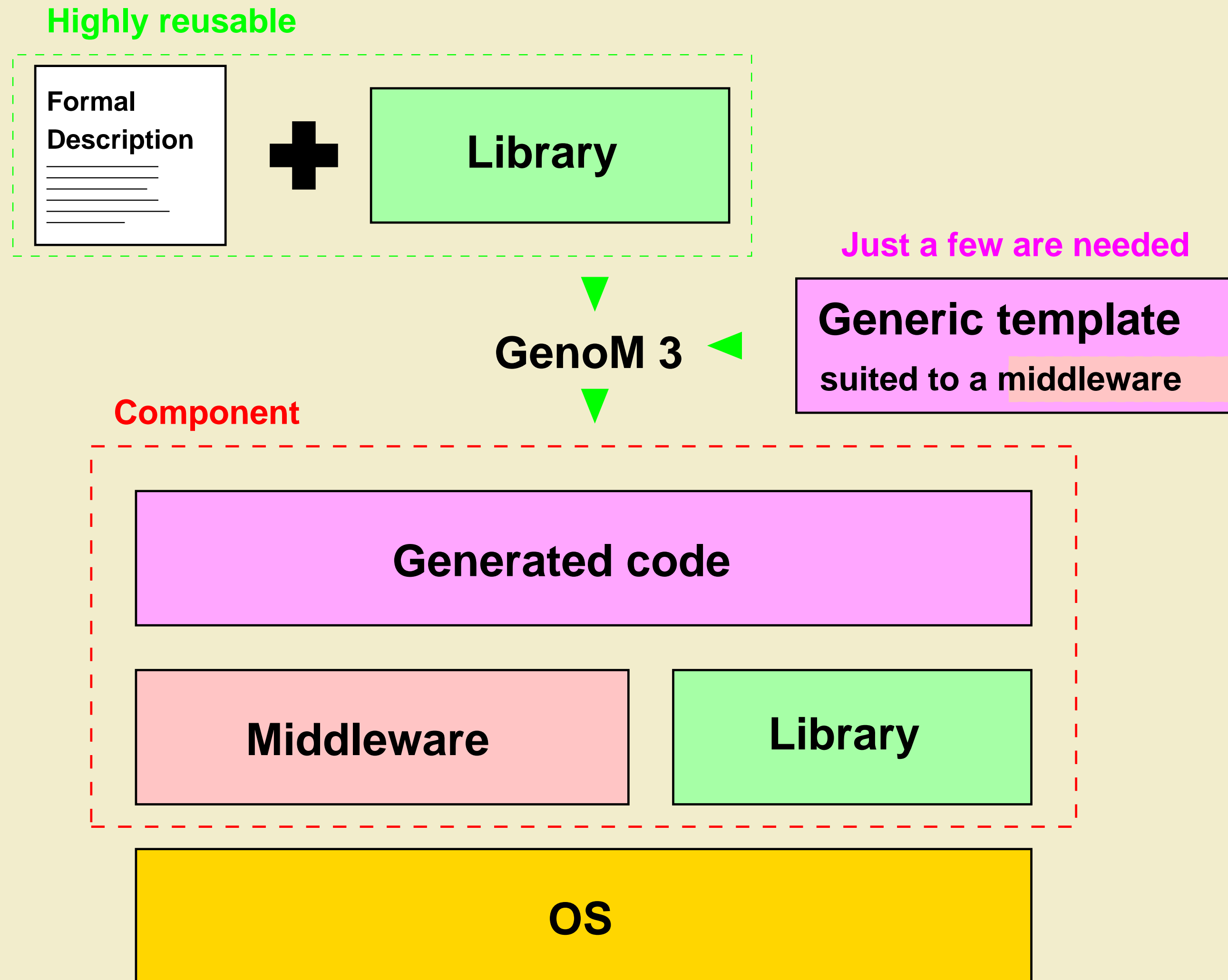


- Components templates define the implementation of the formal description.
- $\longrightarrow$ Code generation.

Key feature of GenoM3: components templates

- Implements the internal machinery of a component.
- Works with an embbeded scripting language (TCL)
  Similar to PHP + HTML web pages
- Can be complex to develop.
  But just **one** template is needed for a given middleware.

General workflow for component generation



- Regular source code with embedded TCL expressions.

```
#include <stdio.h>
<' set component [dogen component] '>

int main()
{
    const char cname[] = "<"$component name">";
    <' foreach s [$components services] { '>
    printf("%s: <"[$s name]">\n", cname);
    <' } '>
    return 0;
}
```

- Master template file: tcl script with helper procedures.
- Main procedure: “template parse”.
- To be called for each template source file.

```
template parse                          \
    args [list arg1 arg2 ...]           \
    file source_filename                \
    file destination_filename
```

- Simply invoke genom3 with template and .gen file

```
# genom3 -h
GenoM 2.99.4 component generator

Usage:
  genom3 [-l] [-h] [--version]
  genom3 [-I dir] [-D macro[=value]] [-E|-n]
         [-v] [-d] file.gen
  genom3 [general options]
         template [template options] file.gen
[...]

# genom3 template component.gen
```

Builtin with GenoM3:

- `skeleton`: generates empty codel skeletons
- `interactive`: launch an interactive TCL
  Useful for template development, debugging, ...

In development:

- `pocolibs`: legacy G^en^oM middleware.
- `bip`: with VERIMAG lab.
- `orocos`: with ONERA lab.
- `ros-comm`: ROS middleware.