Recent evolutions in GenoM

Anthony Mallet, **Matthieu Herrb**

LAAS - CNRS

Grenoble, May 25th, 2011

Overview

1. History and concepts
2. GenoM3
3. Simplified example
4. Internals

Martha EU project – *Circa* 1995

Salient demonstrations

Hilare2 – 2002

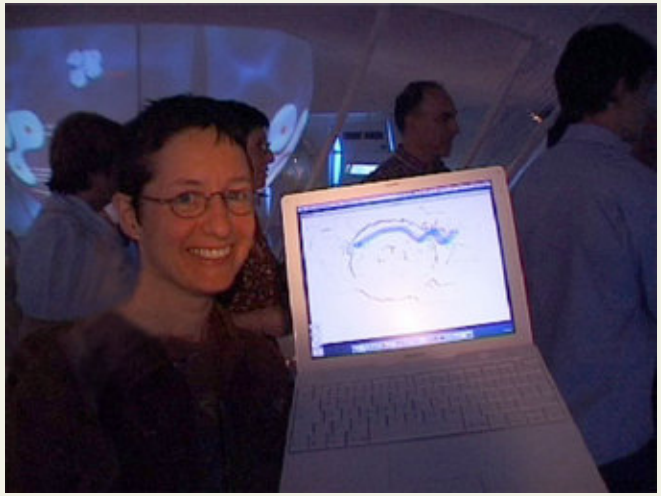
Rackham – 2005

Karma – 2004

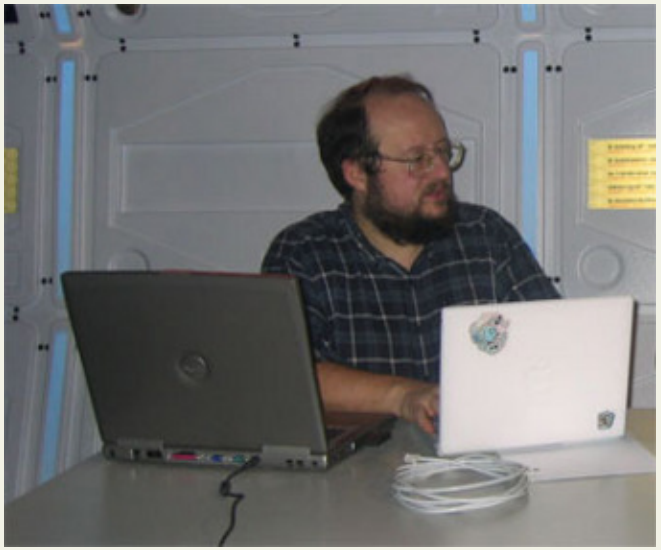
Lama – 2001

Dala – 2004

HRP-2 – 2008



Sara Fleury

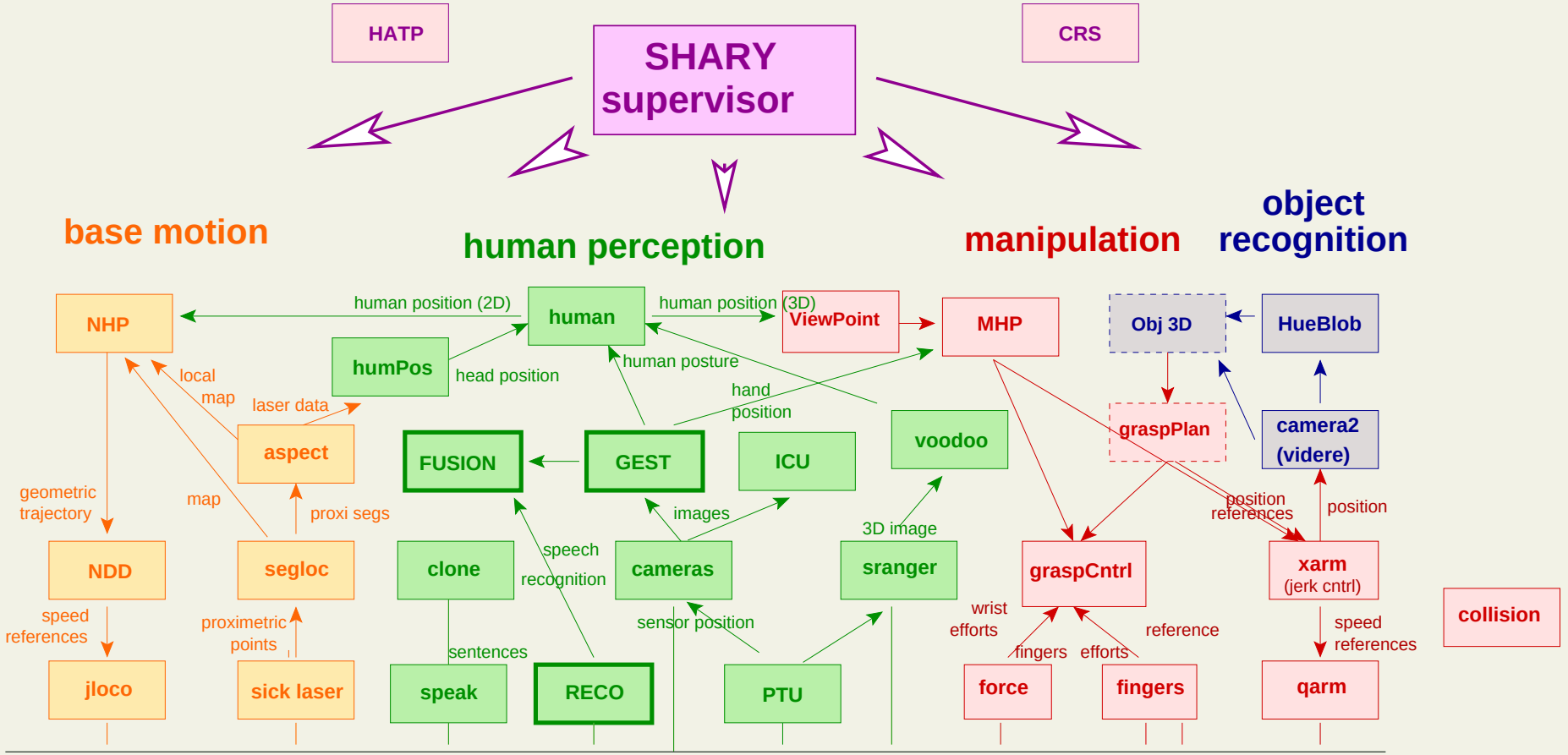


Matthieu Herrb

- 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.

Concepts

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 are defined by a text file:

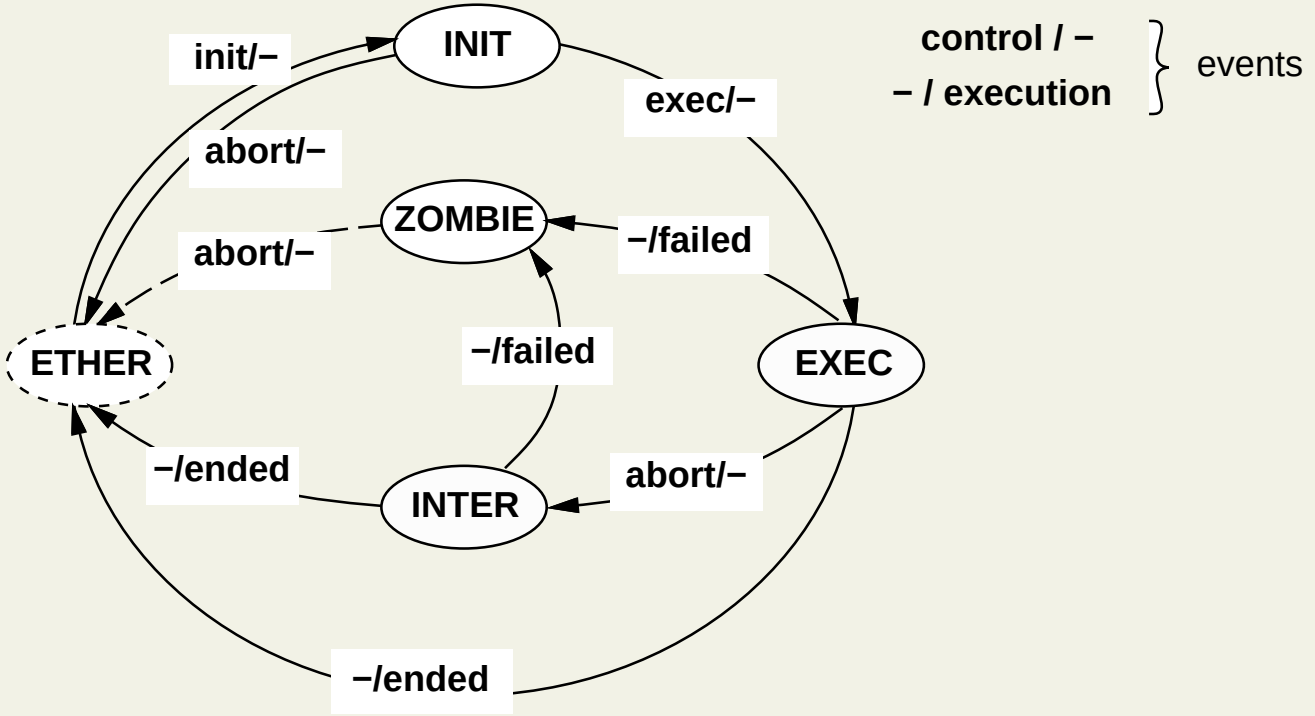
- Interface: ports, services.
- Properties.
- Internals:
  - *codels* (elementary code).
  - *executions contexts* (tasks, threads).

All components share the same component model:

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

- A component defines several services.
- Each running service corresponds to an *activity*.
- G^{en}oM activities execute according to a state machine.

Example: G^{en}oM2 activity FSM.



- 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.

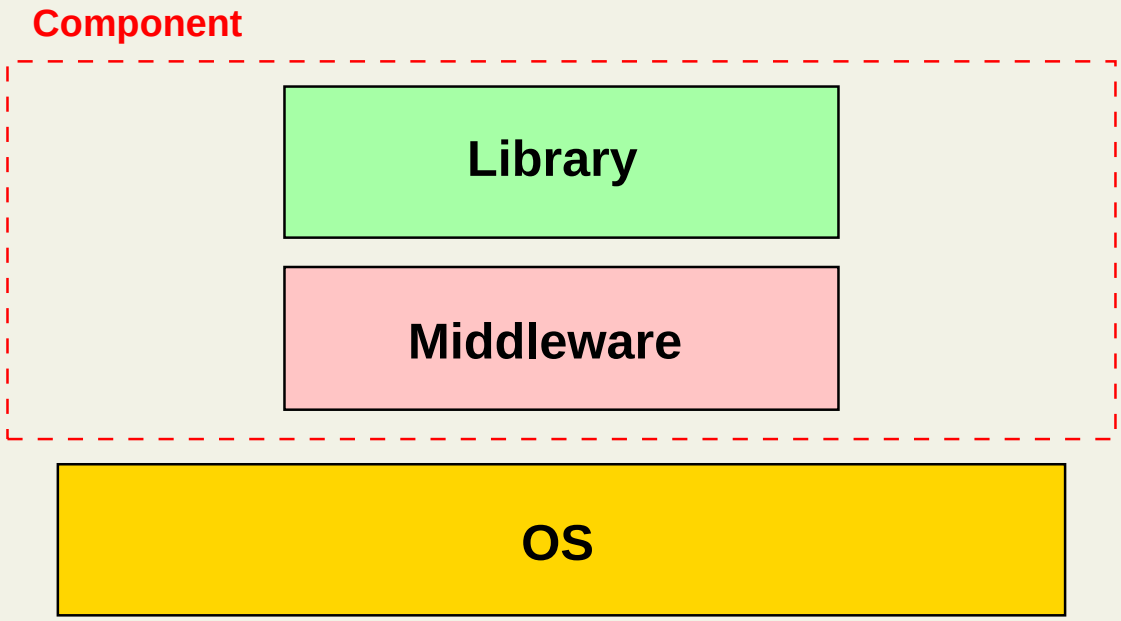
Can we make components truly middleware-independent?

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.

A generic component will usually look like this:



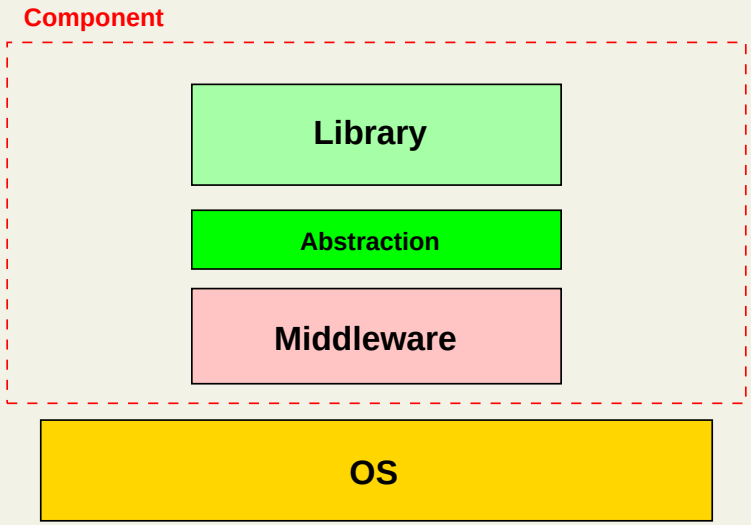
Components:

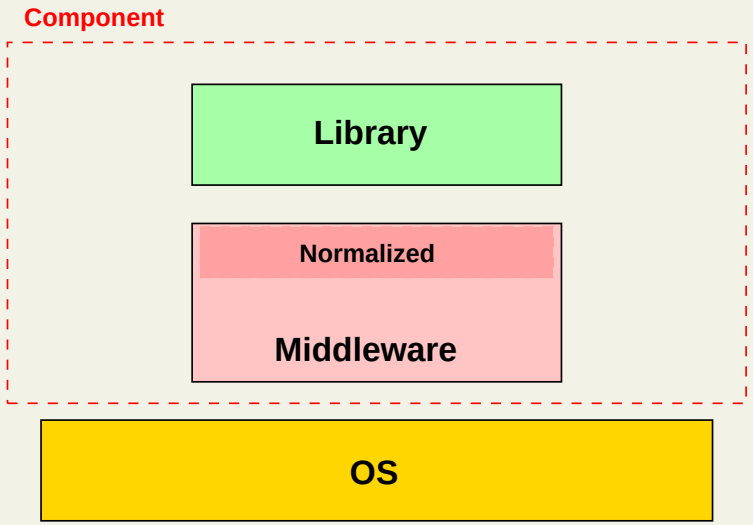
- Be able to use any middleware.
- Make the underlying libraries highly reusable.
- Define a common component model, allowing generic supervision or validation. e.g. BIP from VERIMAG.

Architectures:

- Can be compared (e.g. middleware or architecture benchmarks) in the context of real and complex applications.

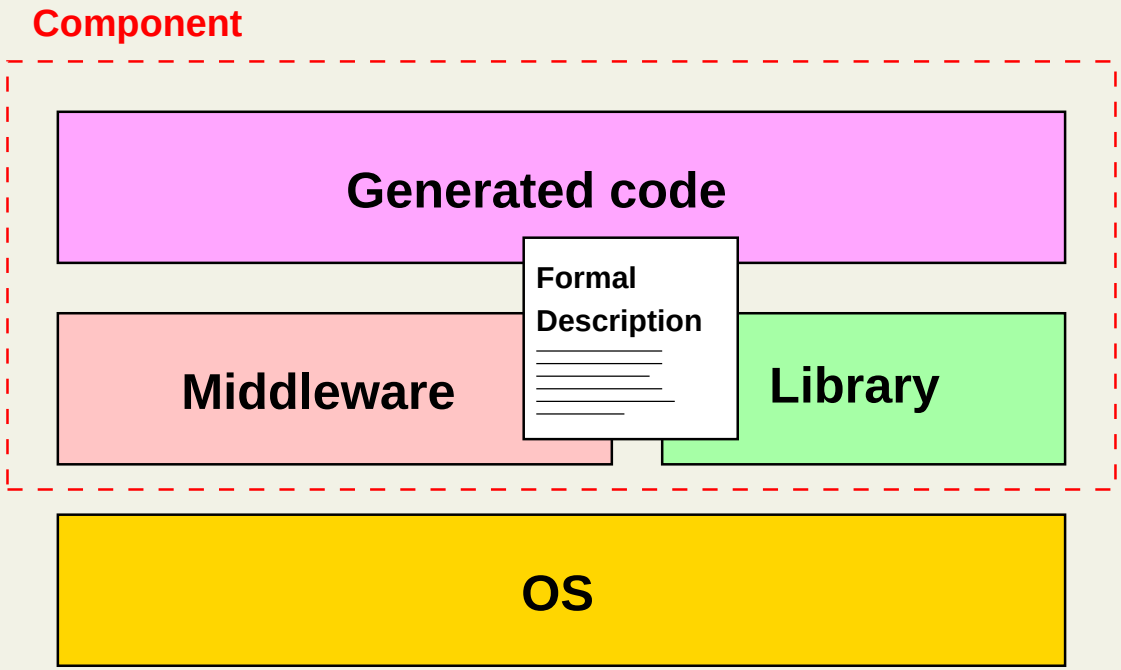
Avoid those approaches:

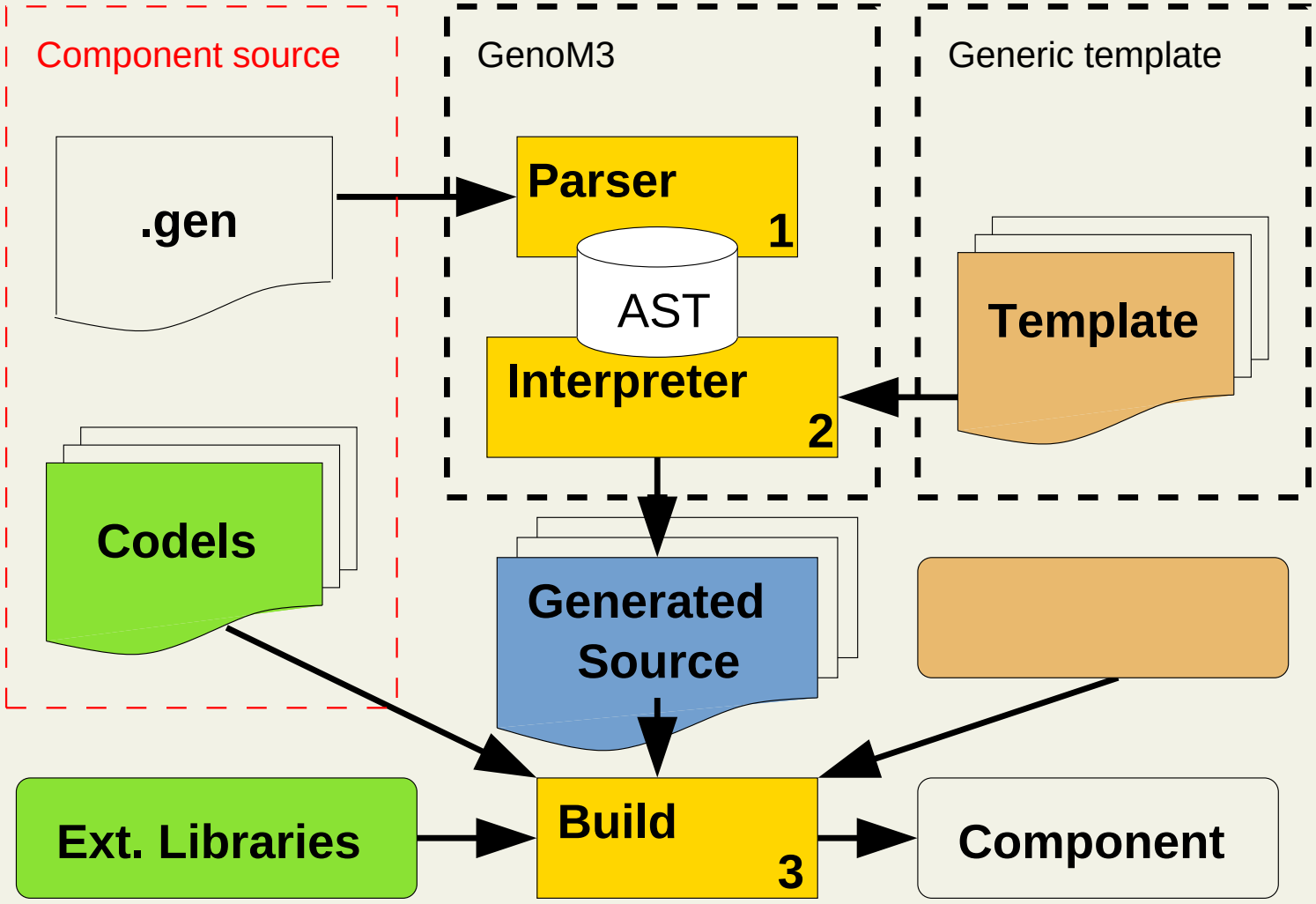




- $\rightarrow$ No component model
  - An extra layer generates extra code and less performance.
  - "Least common denominator" of all the middlewares.
  - Helps a lot for communication, but not for the rest.
    (component interface definition, internal component behaviour, control architecture definition, ...)

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




Component source
GenoM3
Generic template
.gen
Parser
1
AST
Interpreter
2
Template
Codels
Generated
Source
Ext. Libraries
Build
3
Component

- Components templates implement the formal description.
- Templates are tied to the middleware.
- Only one template for a given middleware.

Simplified example: servo control

```
int
validate_k(double *k) {
  if (*k<0.) return ERROR;
  return OK;
}

int
servo_control(double target, double k) {
   double velocity;

   velocity = motor_get_velocity();
   motor_set_velocity(k * (target - velocity));

   return EXEC;
};
```

```
component servo;

ids { double target; double k; };

task main { period: 1ms; }

service set_k(in k) {
  validate: validate_k(inout k);
}

service servo(in target) {
  task: main;
  codel main: servo_control(in target, in k)
      yield main, ether;
};
```

```
% genom3 <template1> <component.gen>
generating ...

% ./configure && make && make install
```

```
% genom3 <template2> <component.gen>
generating ...

% ./configure && make && make install
```

- Formal component description: meta-model
- Templates: component model

- Component interface: IDL data structures

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

- General component description.

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

   ...
};
```

- Internal Data Structure (IDS)
  `ids { ... };`
  - groups all services parameters.
  - state of the component.

- Input/output data ports.

```
inport type name;

outport type 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;
};
```

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.

- Regular source code with embedded TCL expressions.

```
#include <stdio.h>
<’ set component [dotgen component] ’>

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

In development:

- `pocolibs`: legacy G^{en}oM middleware. 95% ready.
- `bip`: with VERIMAG lab?
- `orocos`: with ONERA lab?
- `ros-comm`: ROS middleware?