THE OROCOS PROJECT

Open Robot Control Software

European

Modular Architecture

toward A Specification of Components

Anthony Mallet

February 3, 2002, Pisa (Italy)

CNRS LAAS

Outline

- Description of components

  $\longrightarrow$ Ultimate step until we are able to disseminate code.

- Overview of a generic architecture for control, communication and execution decoupling

  $\longrightarrow$ to understand ideas that are behind the definition of components.

- Examples

  $\longrightarrow$ a motion control framework,

  $\longrightarrow$ an exploration task.

- Components interface specification (overview)

Components and Modules

- **Components:** [...] A software component is an **independent** unit, should be **reusable**, exceeding the concept of “small” entities such as classes, be able to interrogate other components to find their interfaces (and be interrogated itself), be able to generate and handle events [...]. A component should not only document the interface it offers to the world, but also the interfaces it requires from other components in order to do its job.

  $\longrightarrow$ This is what GenoM produces.

  $\longrightarrow$ Should implement the functionalities (path planning, motion control, modeling, ...).

- **Modules:** The unit of software functionality as built in the object-oriented programming paradigm. Basically, this means: data encapsulation and modularization.

  $\longrightarrow$ Generic, modular, reusable libraries.

Components Structure

We not only need to develop functionalities (components), but also *something* that *describes* them (tunable parameters, inputs, outputs, time properties, ...), *i.e.* a **description language** for robotic functionalities. (roughly the same idea as *e.g.* IDL and Corba).

Proposition: define components that are made of (at least)

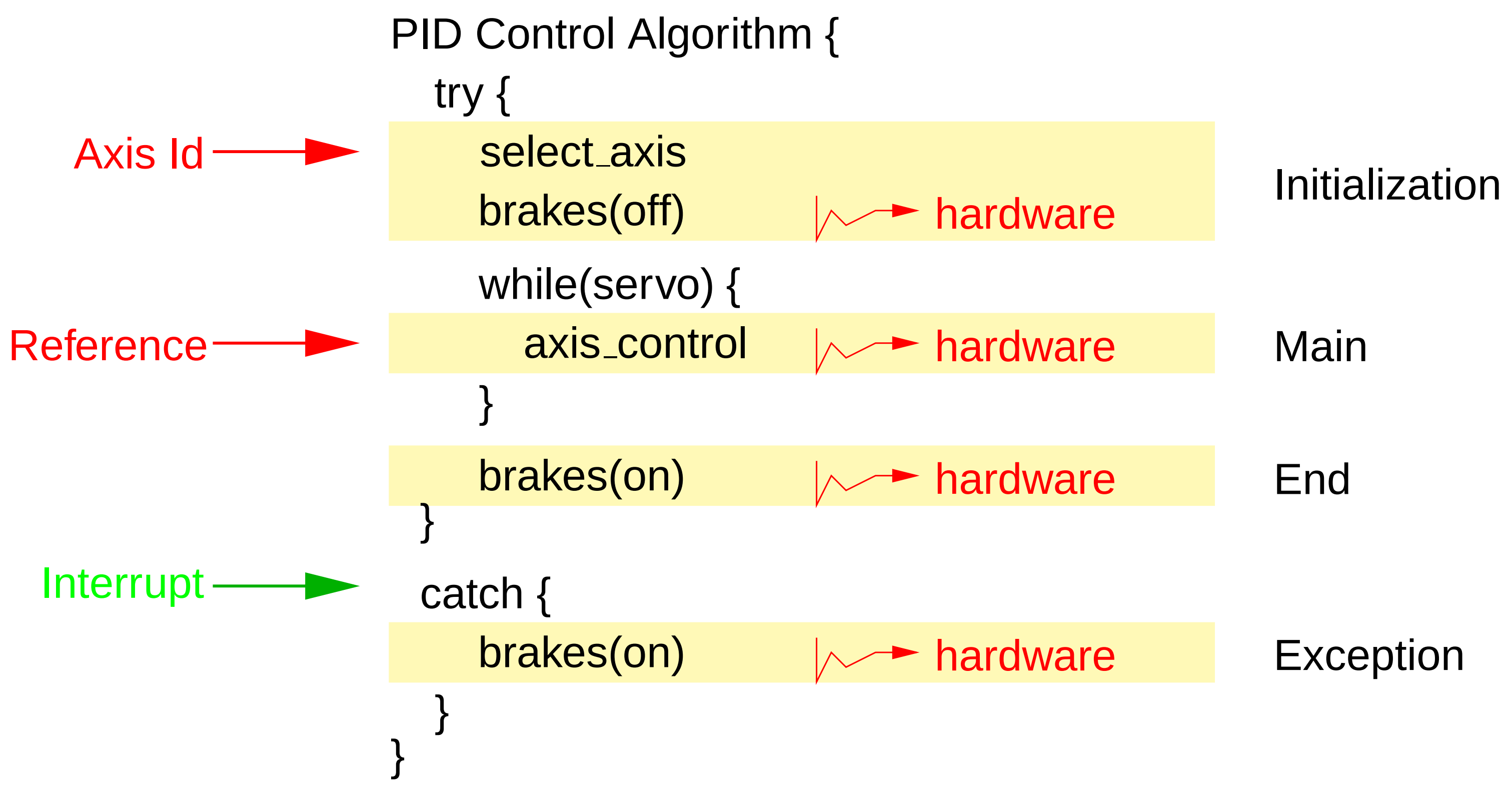
- a set of codels,
- an execution engine,
- communication libraries.

The talk will show how this can answer our needs in terms of modularity, reusability, (re)configurability, ...

Codels

- Basically, codels are *functions* (or methods of a class) that **structure** an algorithm into different parts. They are written by the developper of a particular functionality in order to implement the core of a component.
- Their execution is sequenced *elsewhere*.
- They do not handle the communication with other components.
- They are **atomic**. They cannot be interrupted.
- They are especially well suited for periodic execution.

Example of Codels

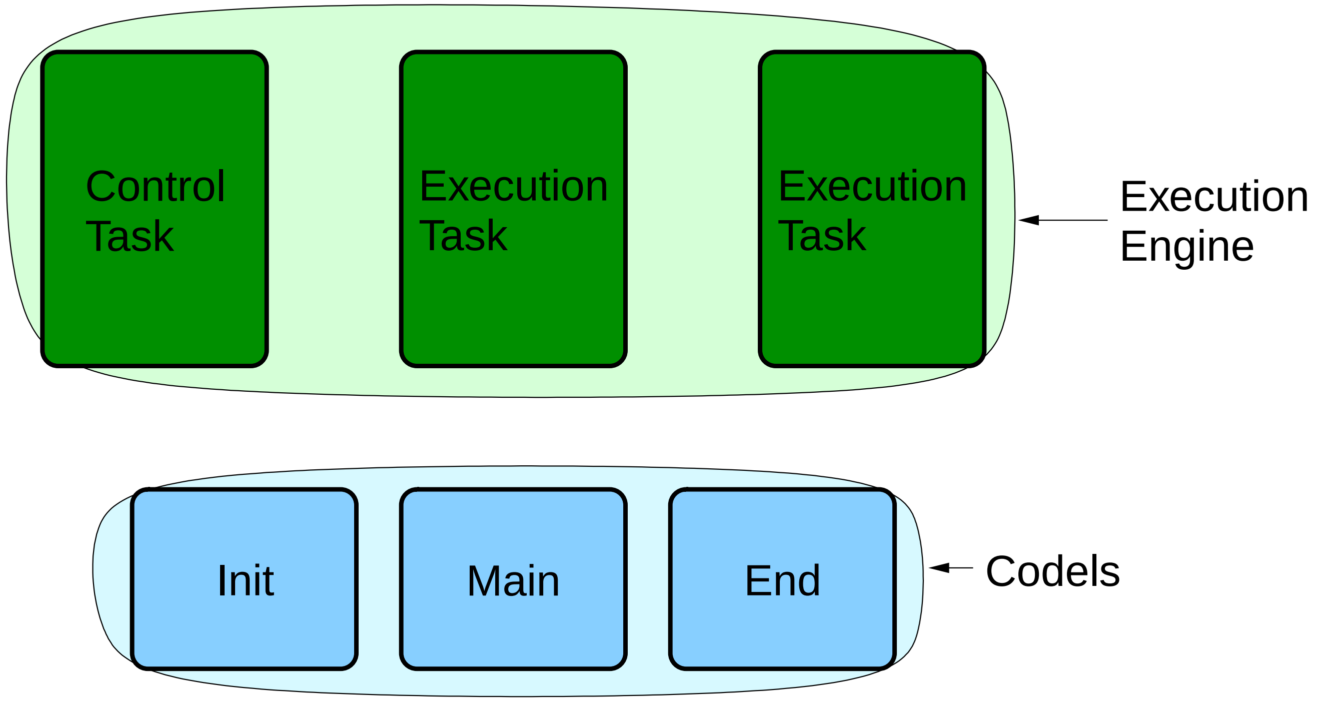


```
PID Control Algorithm {
   try {
      select_axis
      brakes(off)
      while(servo) {
         axis_control
      }
      brakes(on)
   }
  catch {
      brakes(on)
   }
}
```

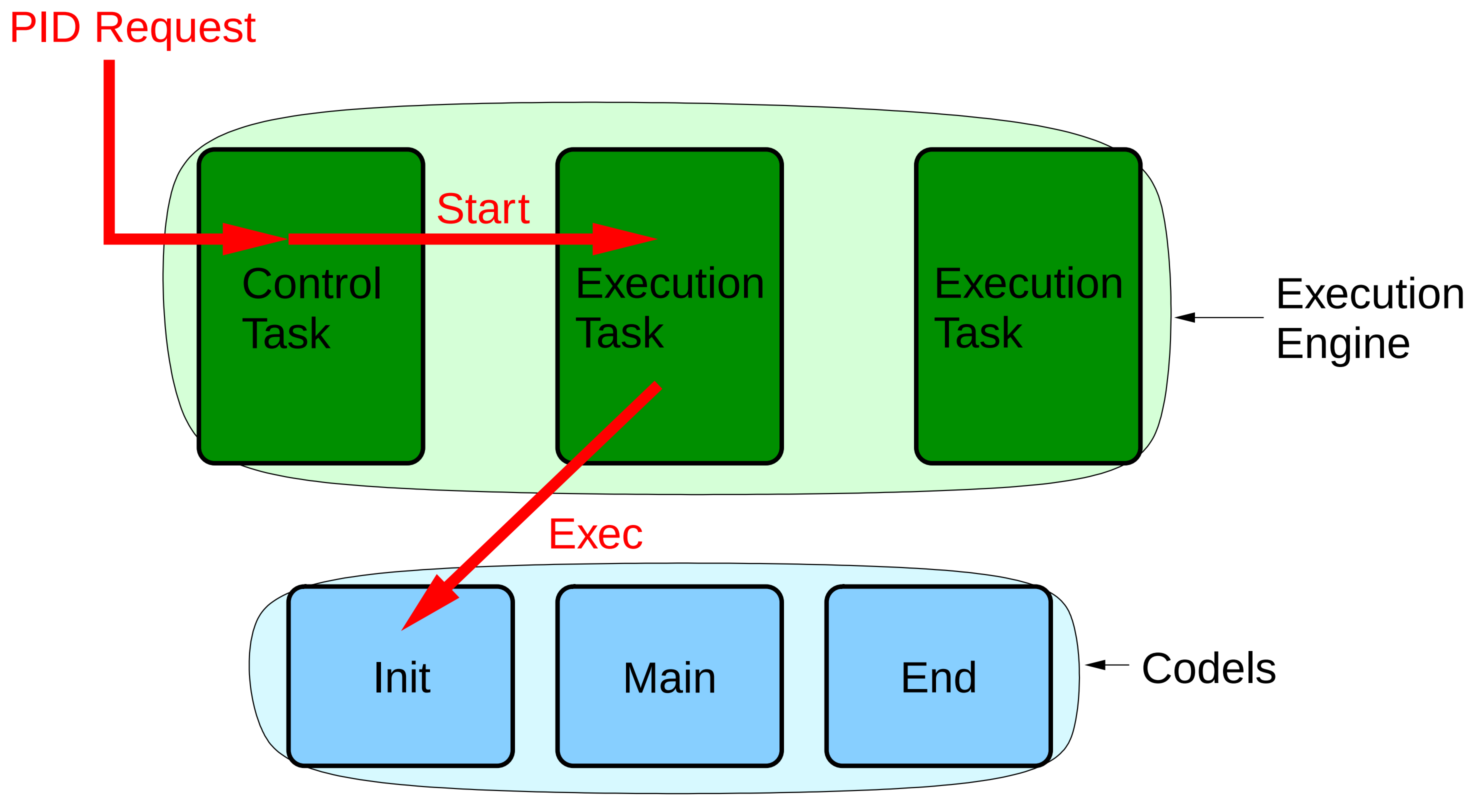
Execution Engine

- Execution engine **sequences the codels** execution and **handles communication** between components.
- It **provides services** which are made available to the outside of the component. Services correspond to a sequence of codels.
- It is **generic**, and written once by the developpers of the *system* (reusable between components).
- There can be different kind of engines, with different properties (real-time) or different execution model (FSM, Petri nets, ...).
- An execution engine properly linked with a library of codels makes an executable (and a component).

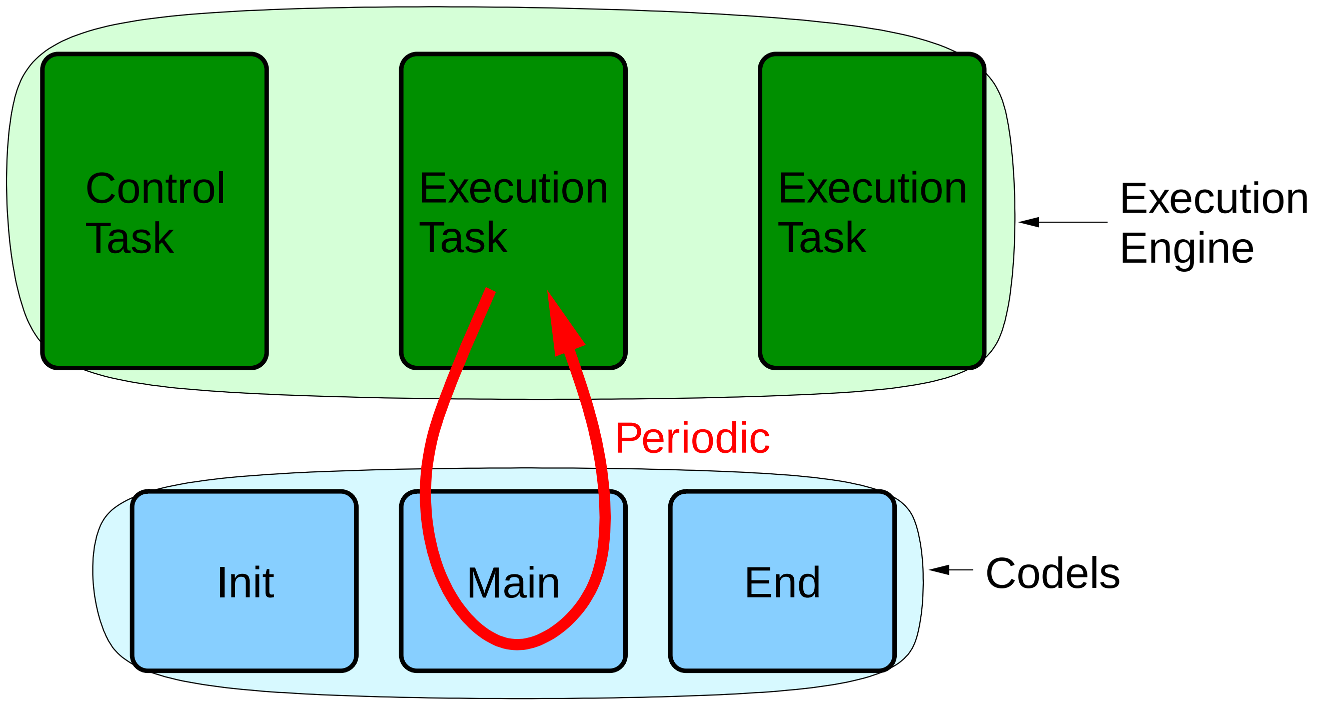
Execution Example



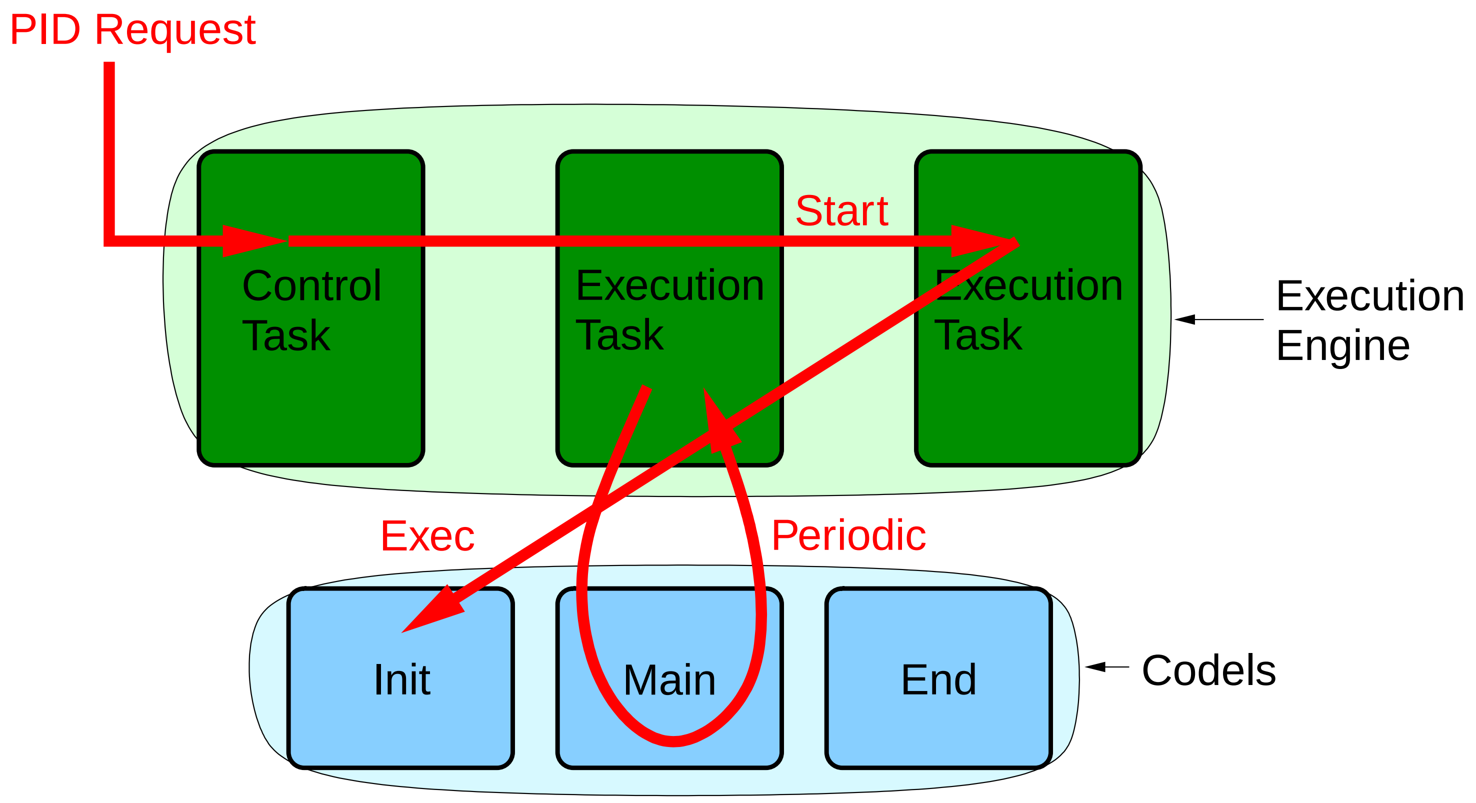
Components Structure — Execution Example



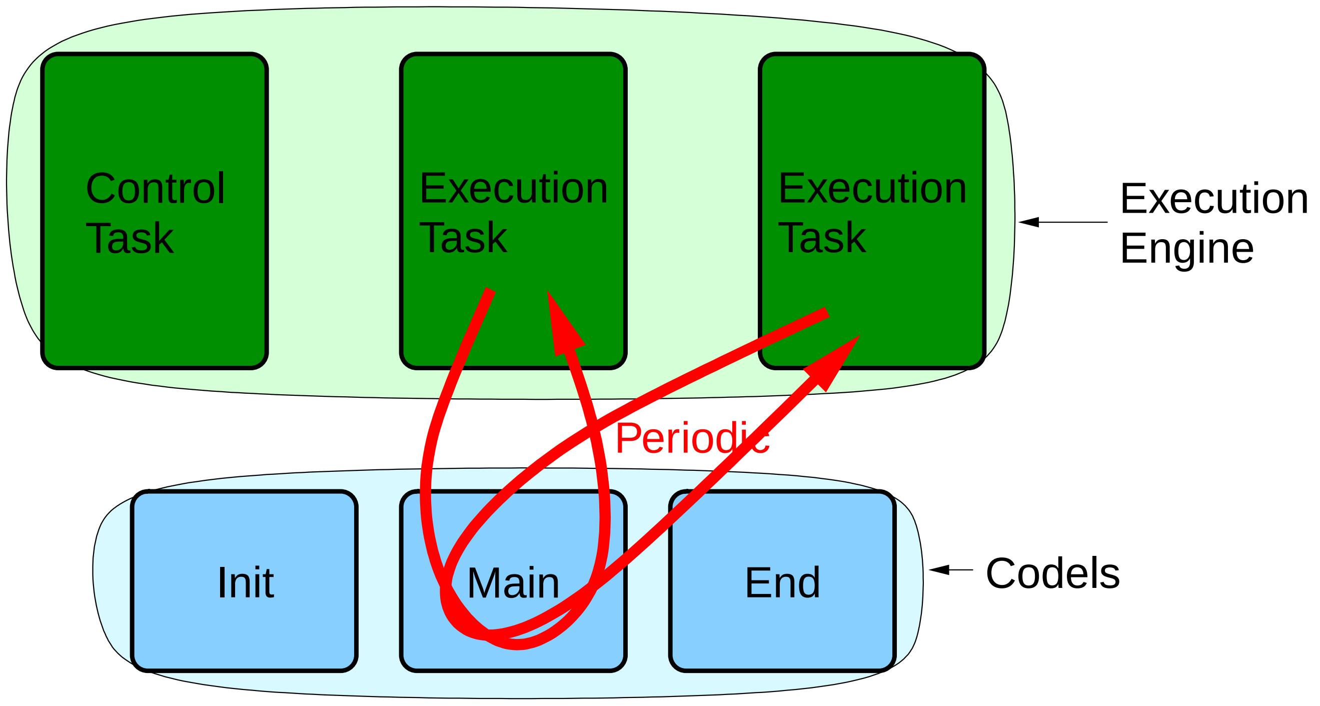
Execution Example



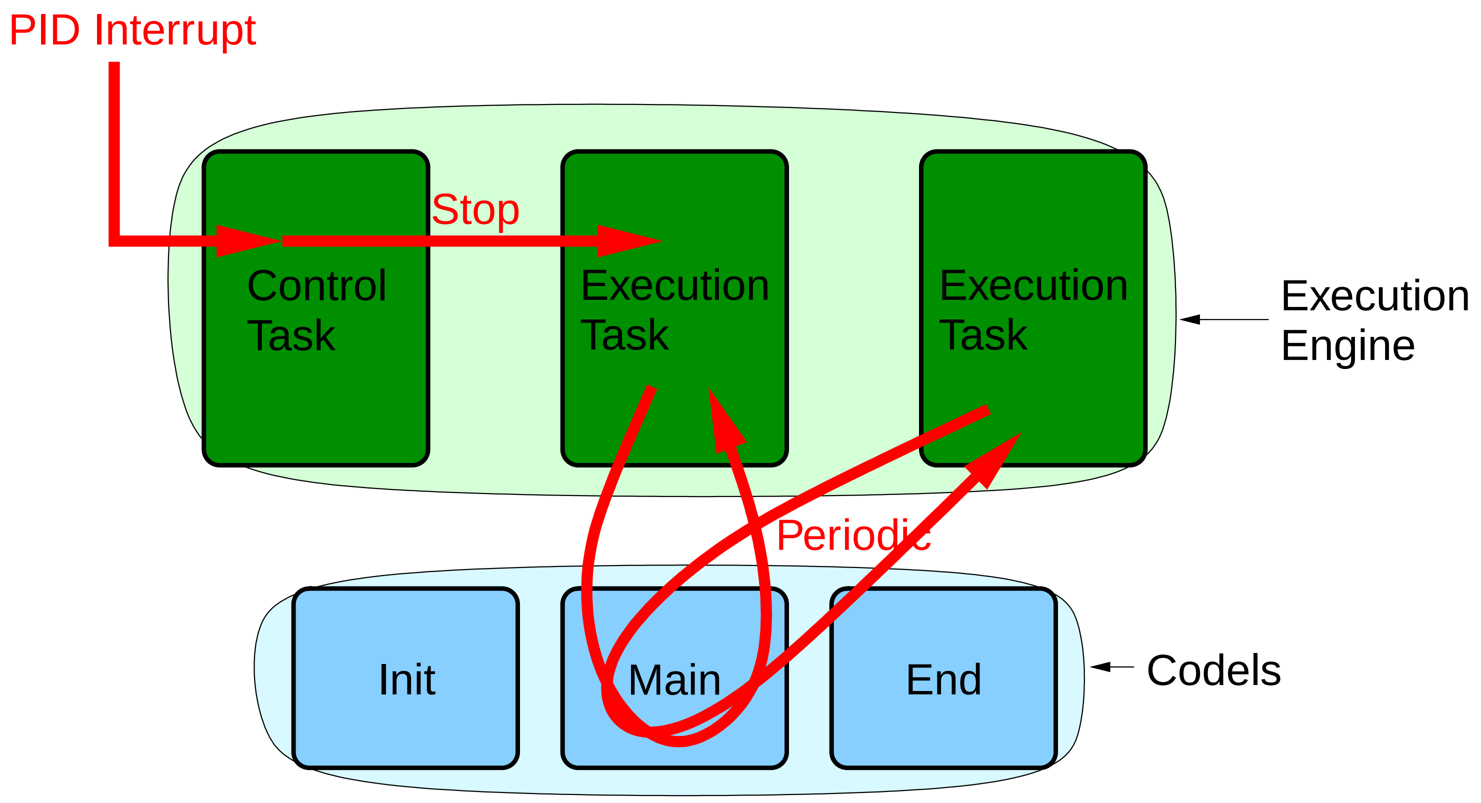
Execution Example



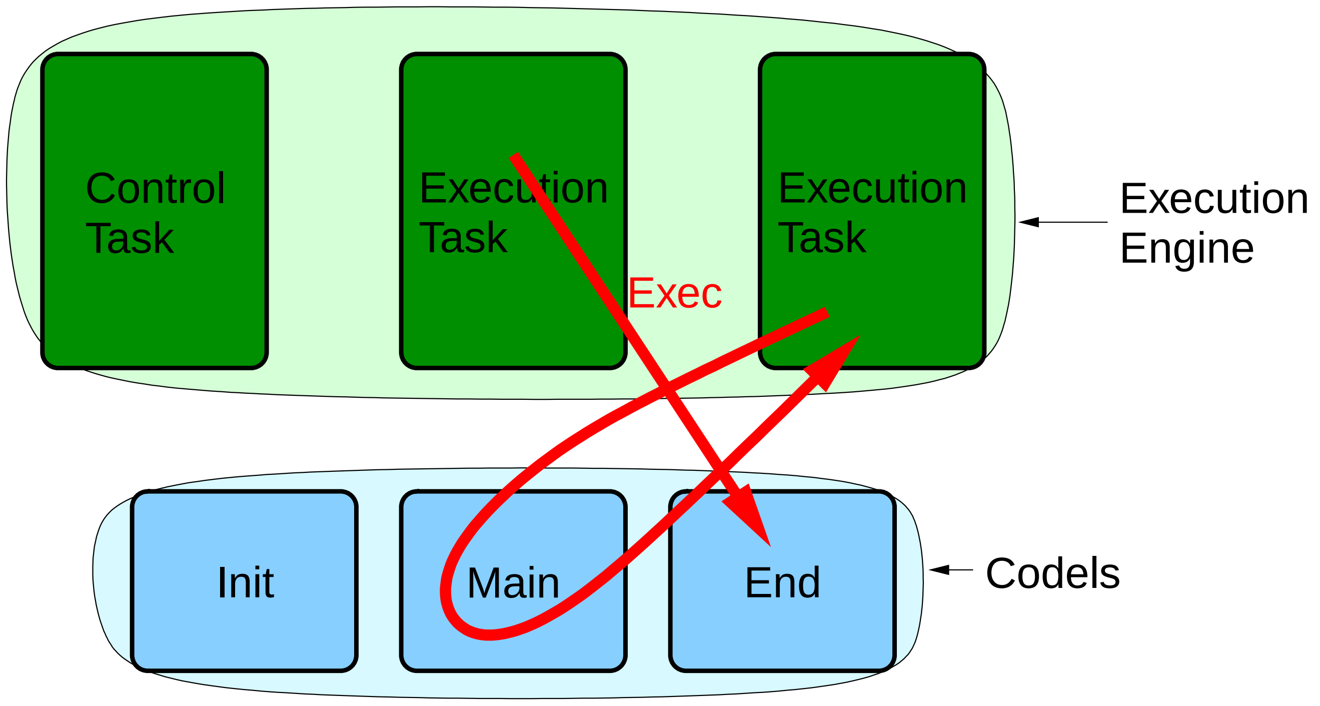
Execution Example



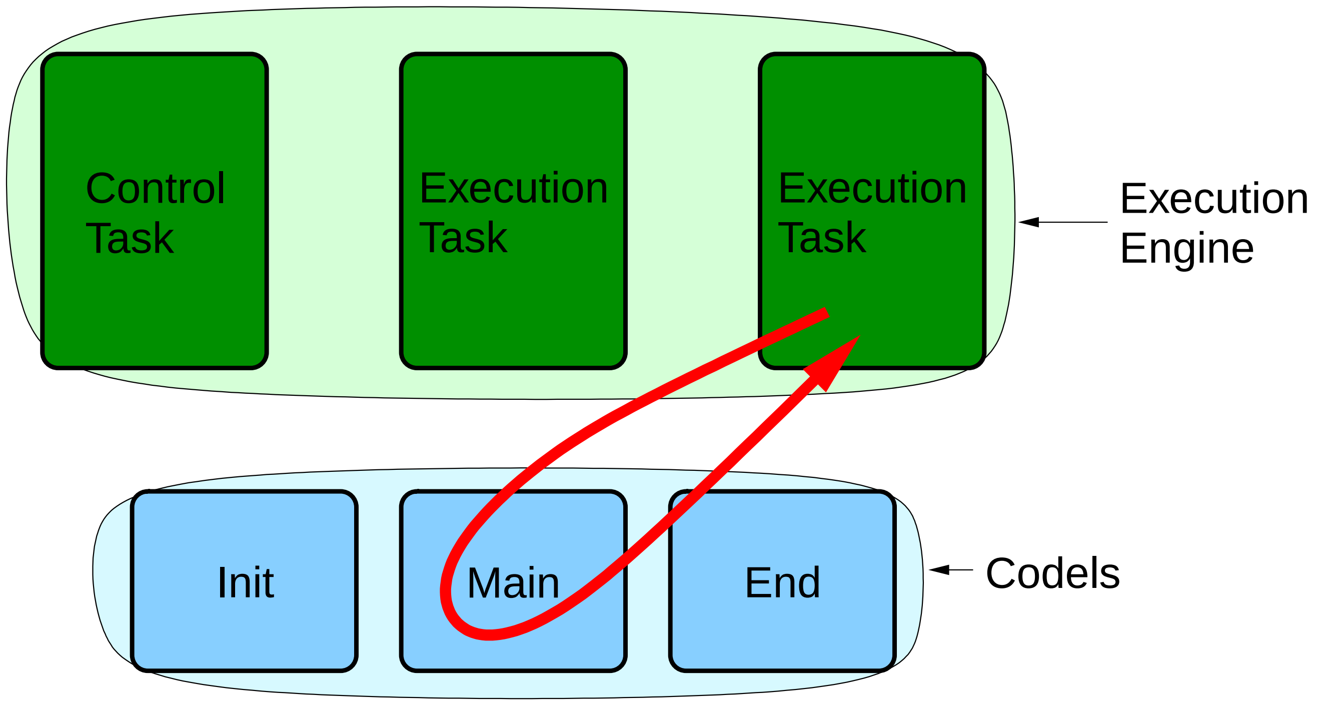
Execution Example



Execution Example



Execution Example



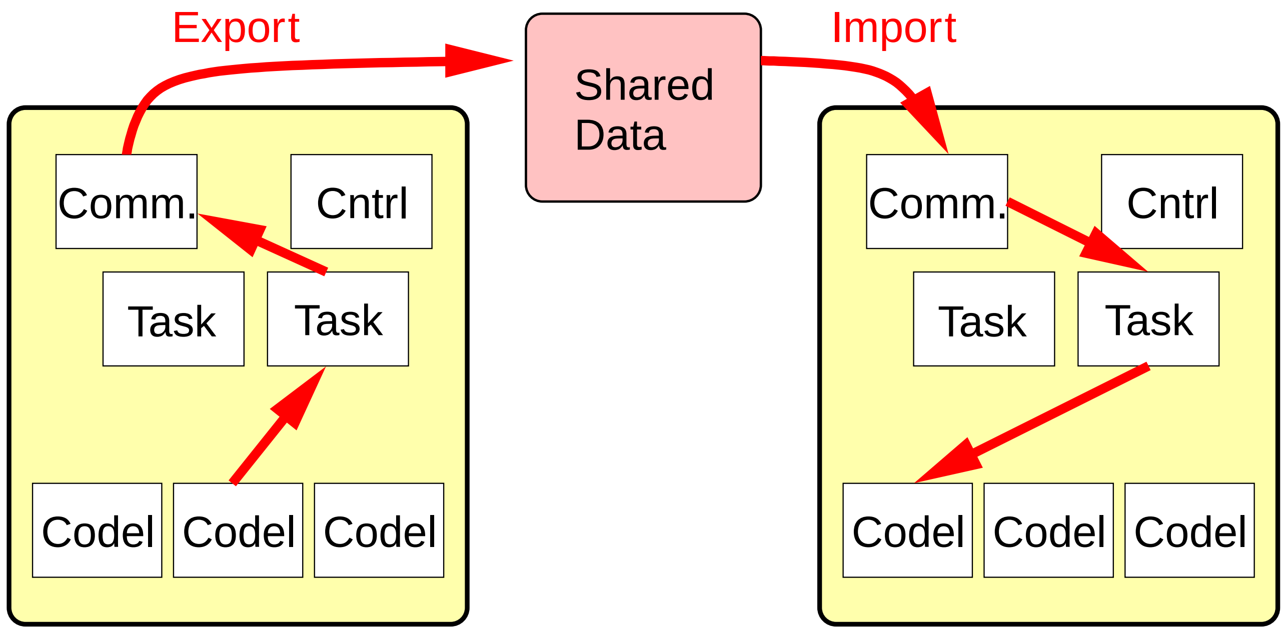
$\longrightarrow$ One single task can (could) process several requests! (not shown here).

Exported and Imported Data

- Codels can use data coming from (and produce data for) other components.
- Codels do not take care of this and only define their interface.
- Before running a codel, the execution engine fetches what the codel need and passes the data to it. After the codel has executed, data can also be exported by the engine in a symetrical way.

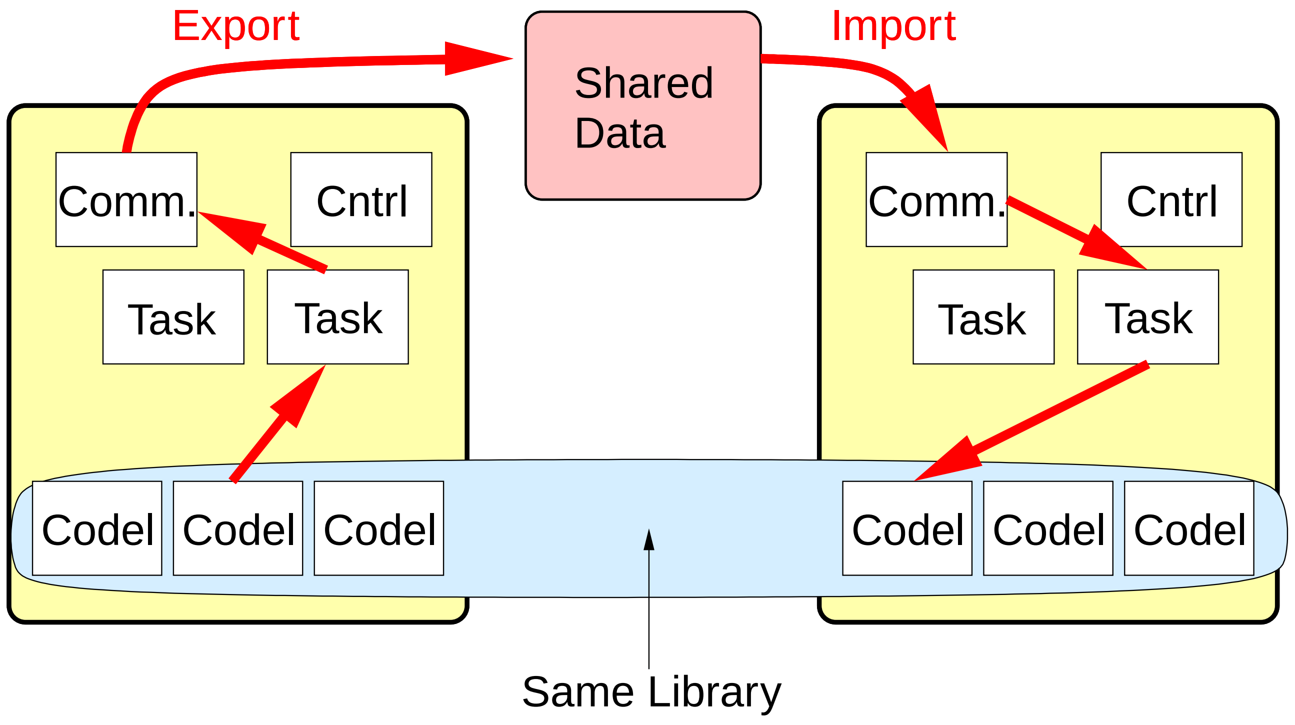
$\longrightarrow$ Communication is typically encapsulated into a *library* (module?) which is used by the execution engine (thus allowing different protocols / implementations /...).

Exported and Imported Data

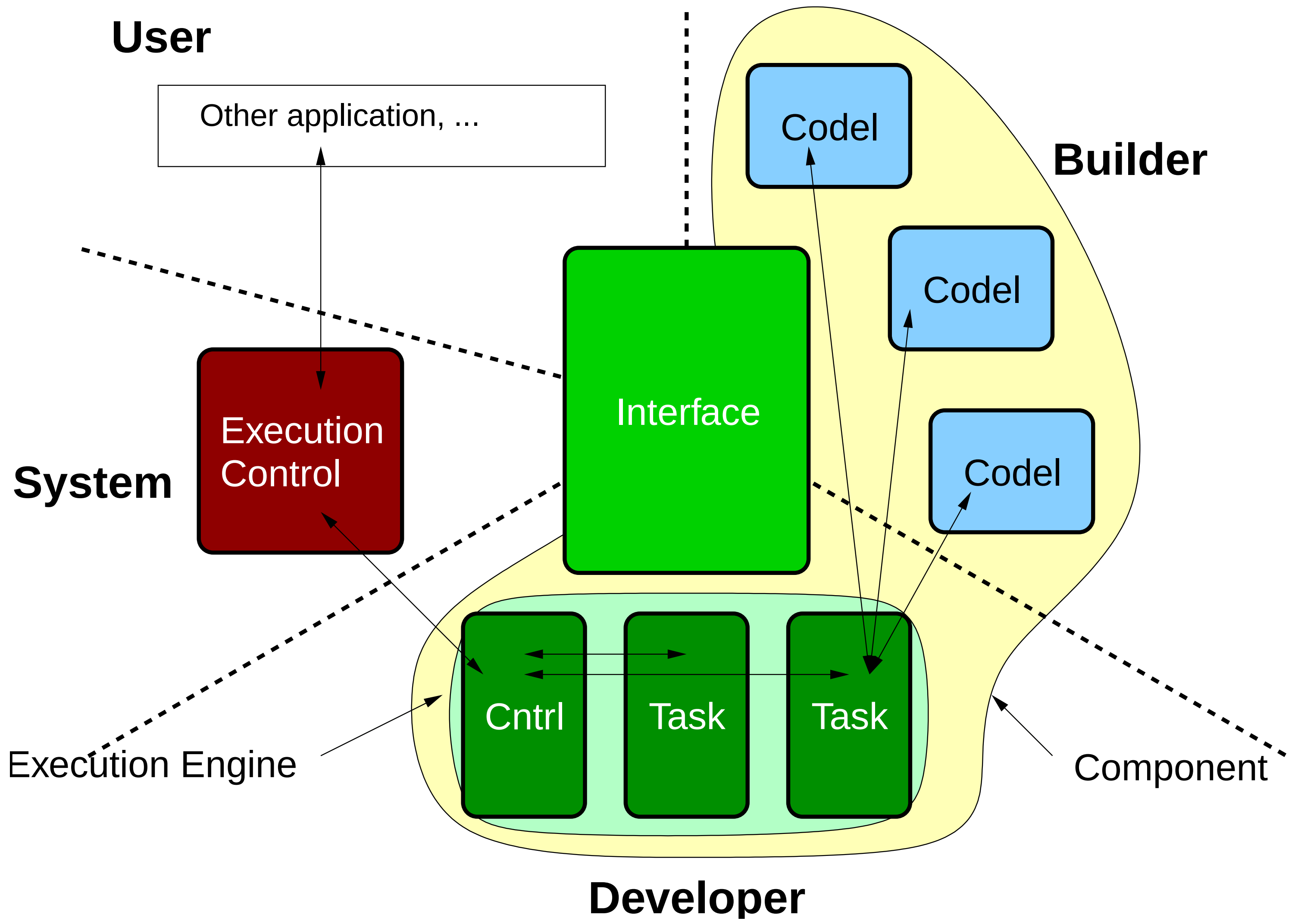


Exported and Imported Data

- Also compatible with an object-oriented approach (*e.g.* codels in different components can share the same libraries).



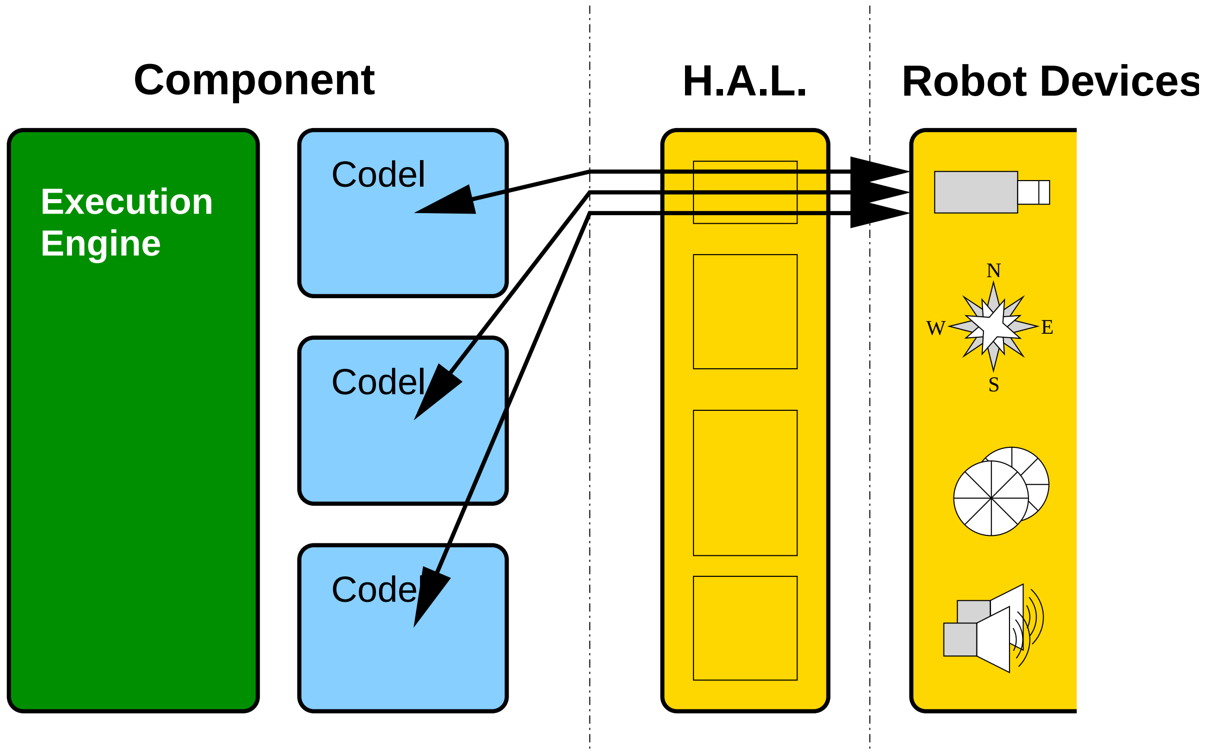
Overview


User
Other application, ...
Builder
Codel
Codel
Codel
Interface
Execution Control
System
Cntrl
Task
Task
Execution Engine
Component
Developer

Hardware Abstraction

- Some components will act as devices abstractions (*e.g.* cameras, sonars, motors, ...)

  $\longrightarrow$ Abstraction is done by codels through the H.A.L.



Control and Data Flows

- Control Flow

  → Setting parameters, starting or ending executions, ...

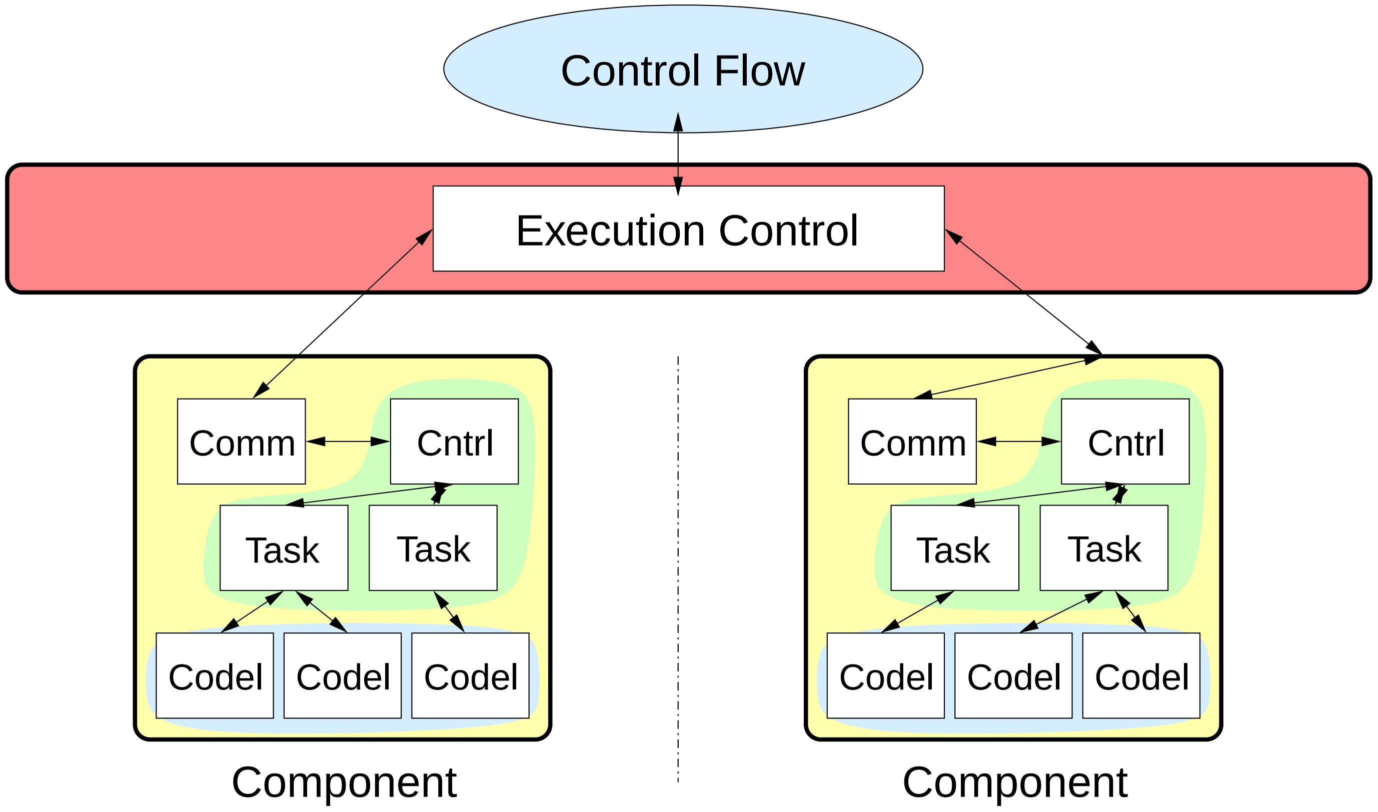
- Data Flow

  → Data transfers (between components) during execution. (images, sonar echoes, positions, ...).

Control flow and Data flow must be decoupled.

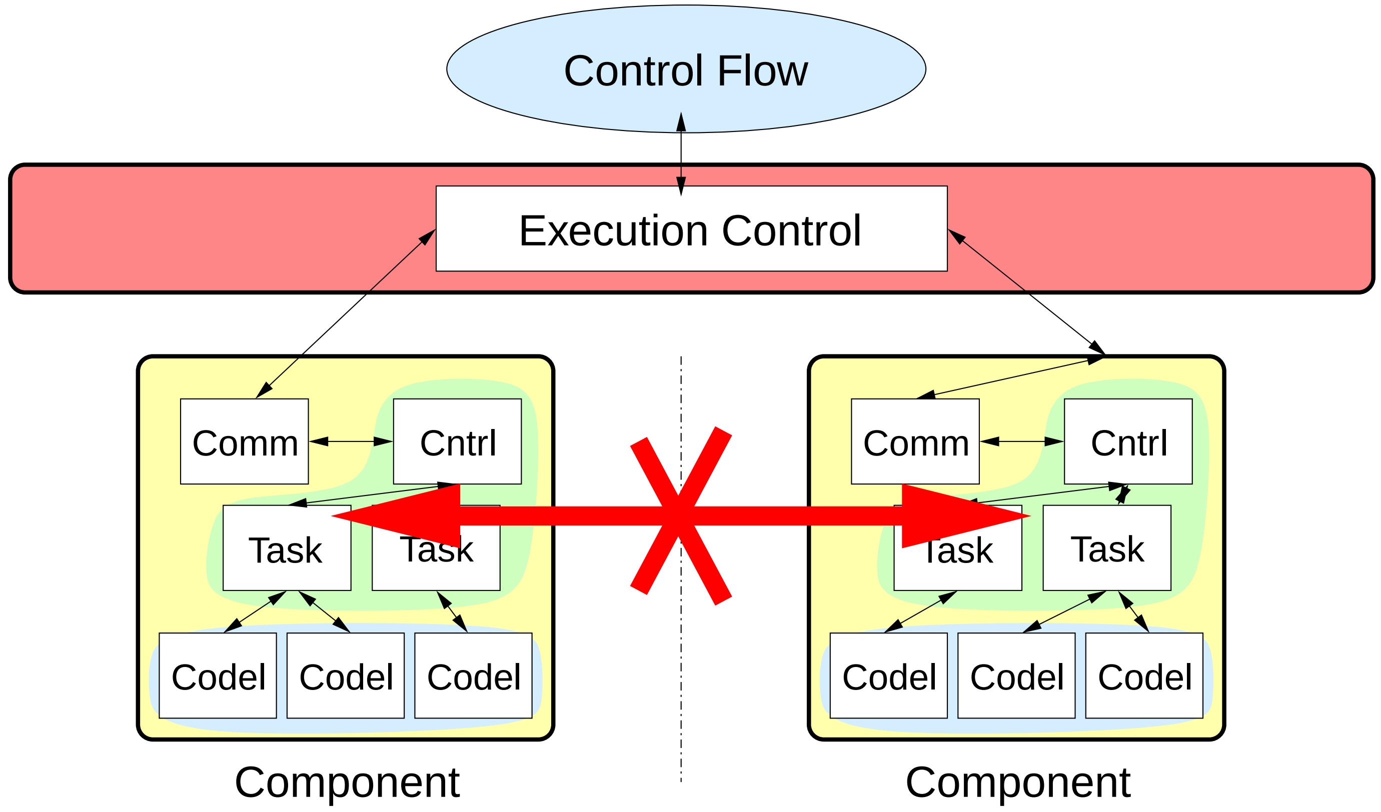
Control Flow

- The control flow has to be defined outside components, because a component does not have the necessary knowledge to do this.



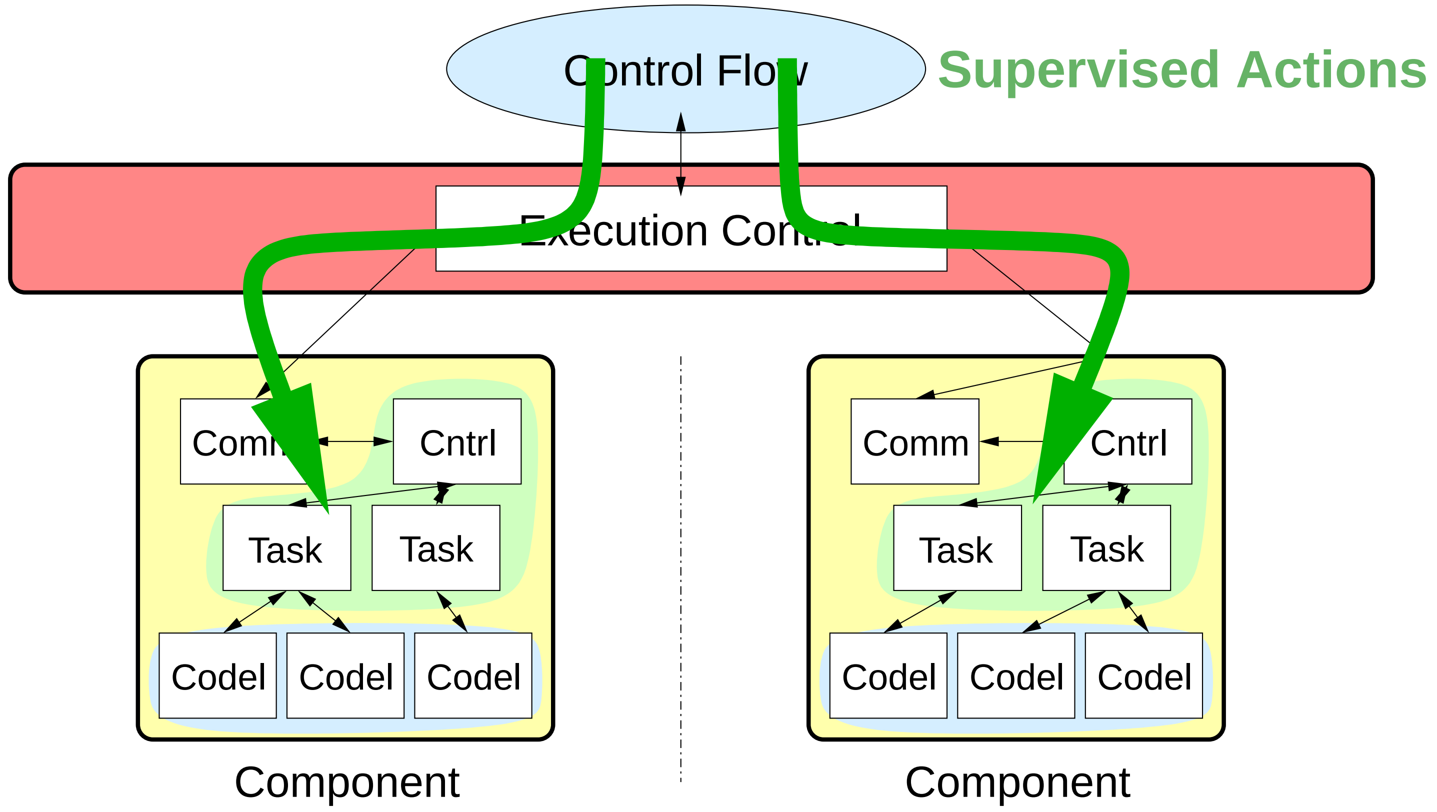
Control Flow

- The control flow has to be defined outside components, because a component does not have the necessary knowledge to do this.



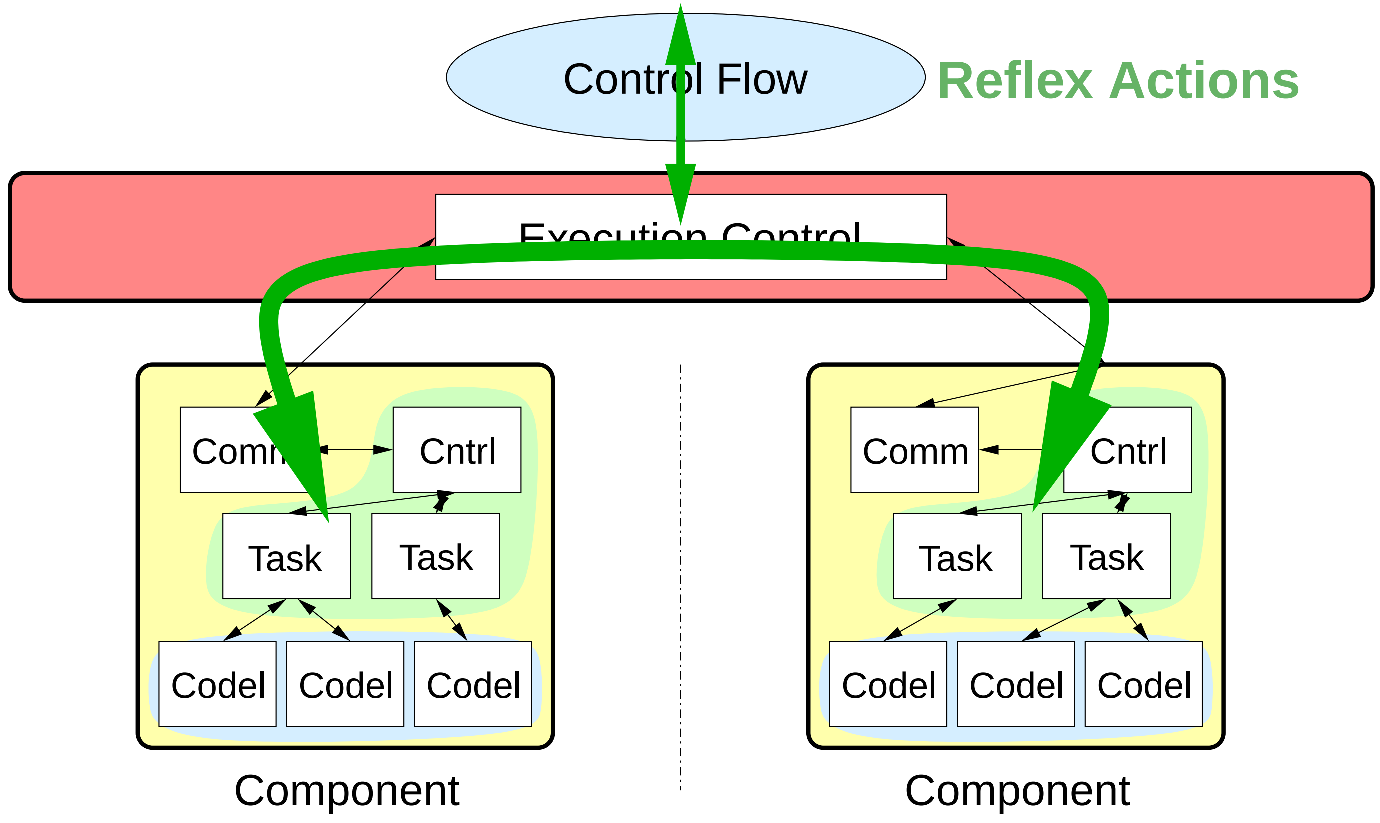
Control Flow

- The control flow has to be defined outside components, because a component does not have the necessary knowledge to do this.

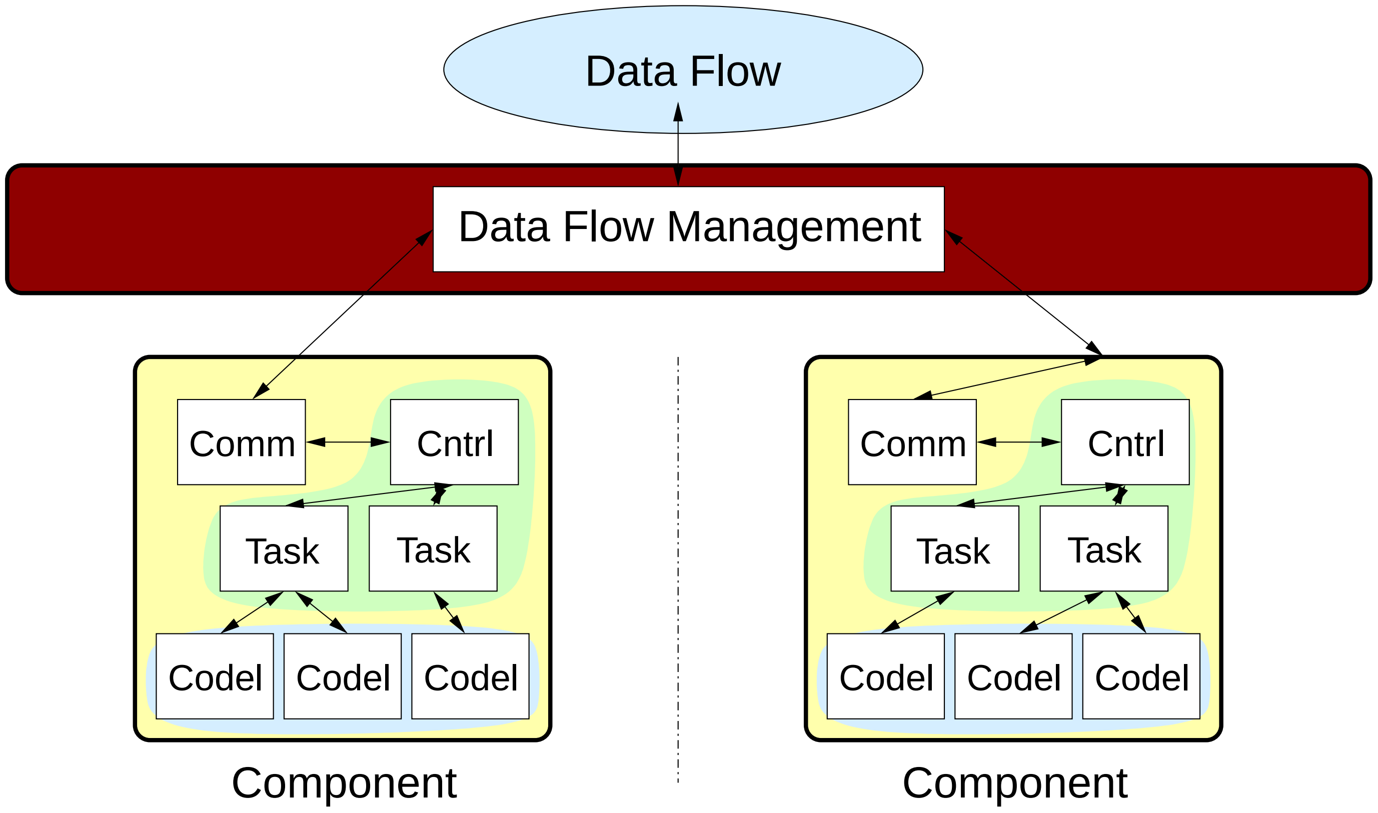


Control Flow

- The control flow has to be defined outside components, because a component does not have the necessary knowledge to do this.

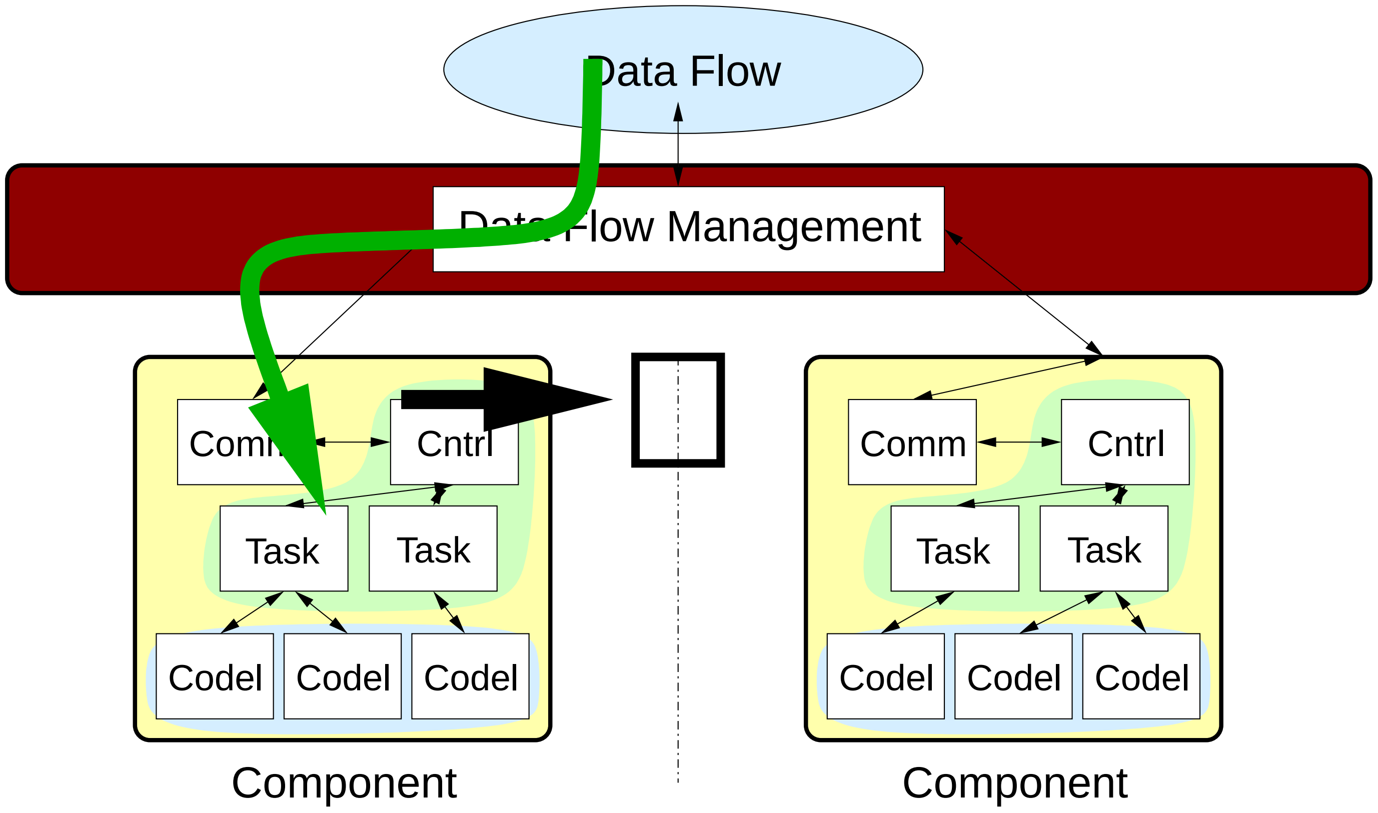


Data Flow



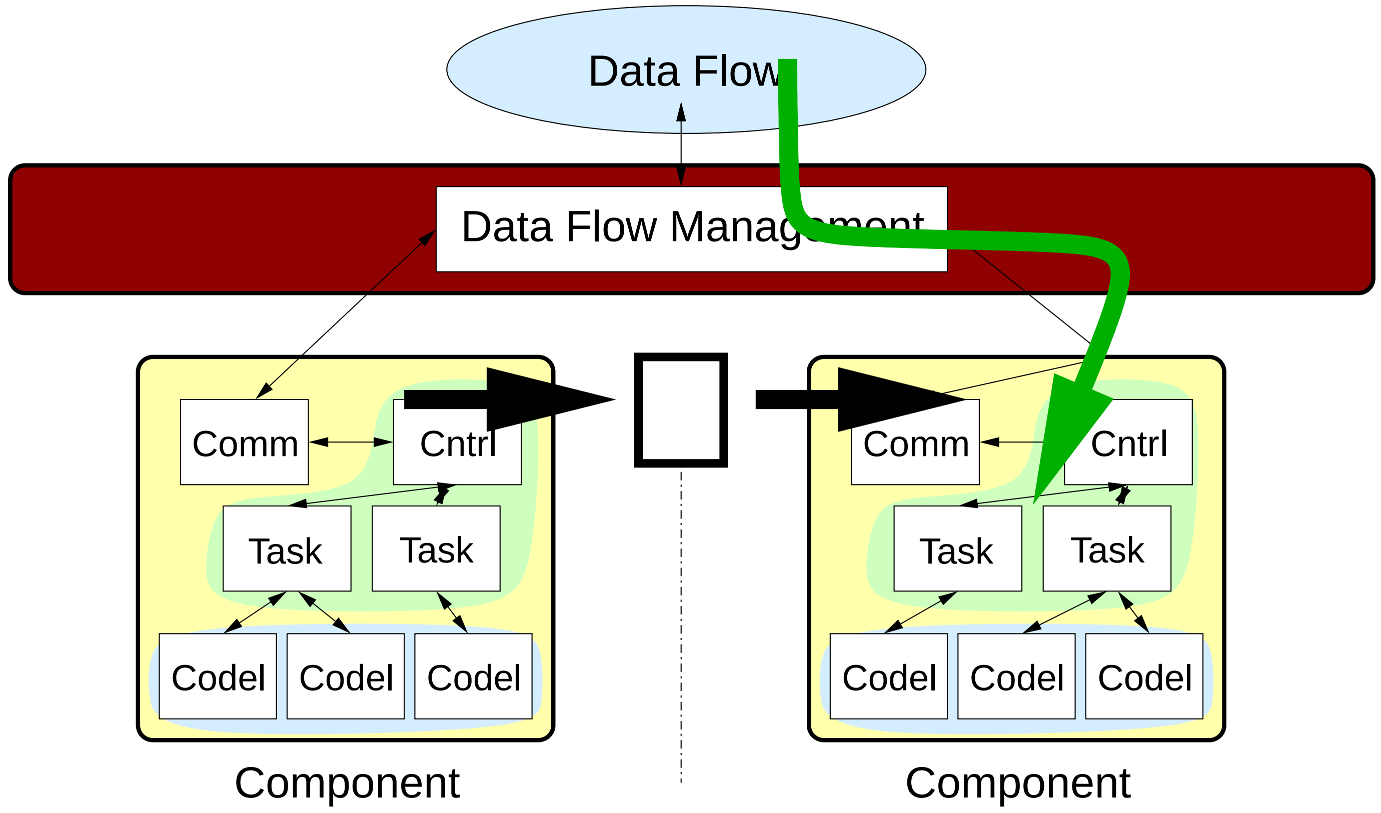
- The data flow should also be defined outside components and should be **decoupled** of the control flow. It should also be **controlable**.

Data Flow



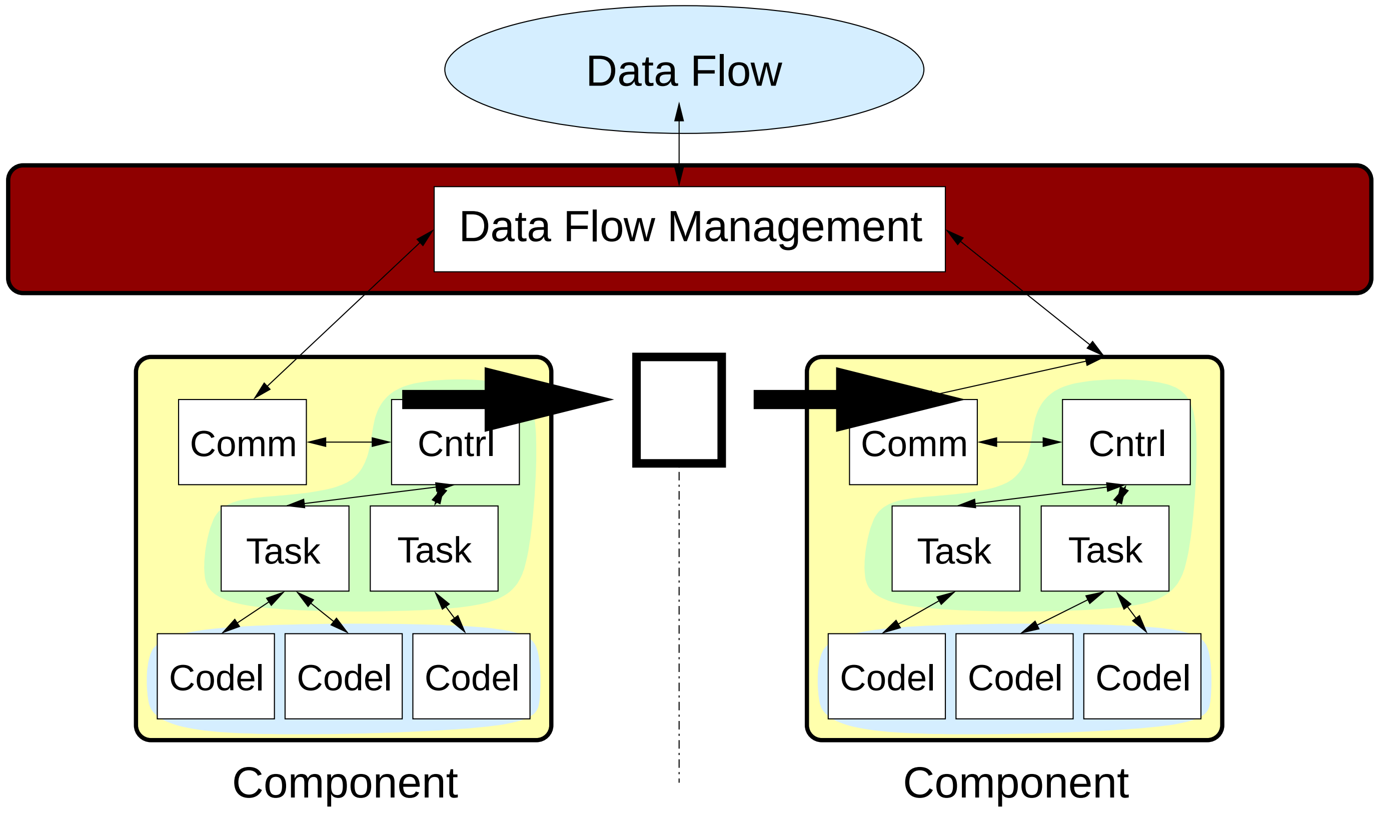
- The data flow should also be defined outside components and should be **decoupled** of the control flow. It should also be **controlable**.

Data Flow



- The data flow should also be defined outside components and should be **decoupled** of the control flow. It should also be **controlable**.

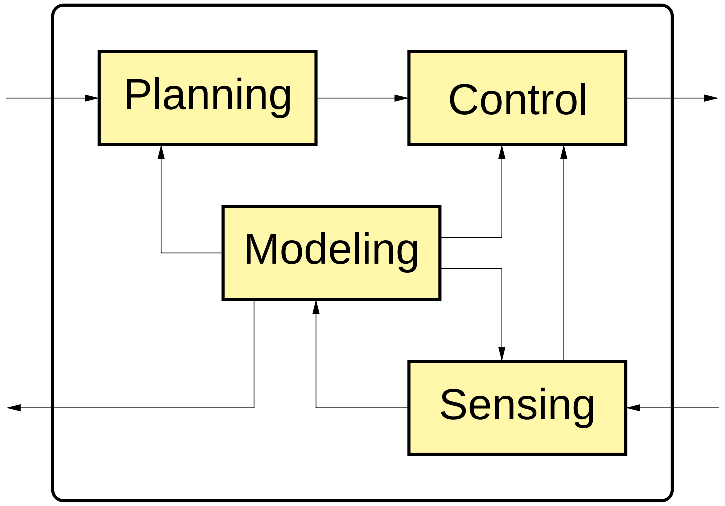
Data Flow



- The data flow should also be defined outside components and should be **decoupled** of the control flow. It should also be **controlable**.

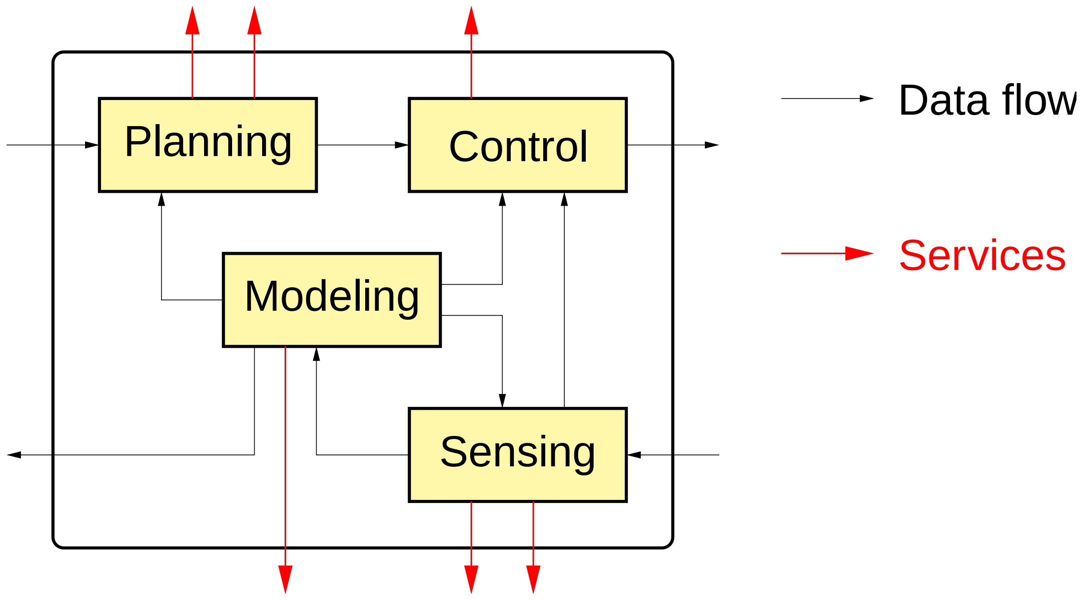
Motion control

- Motion control framework. All the yellow boxes could be implemented as components.



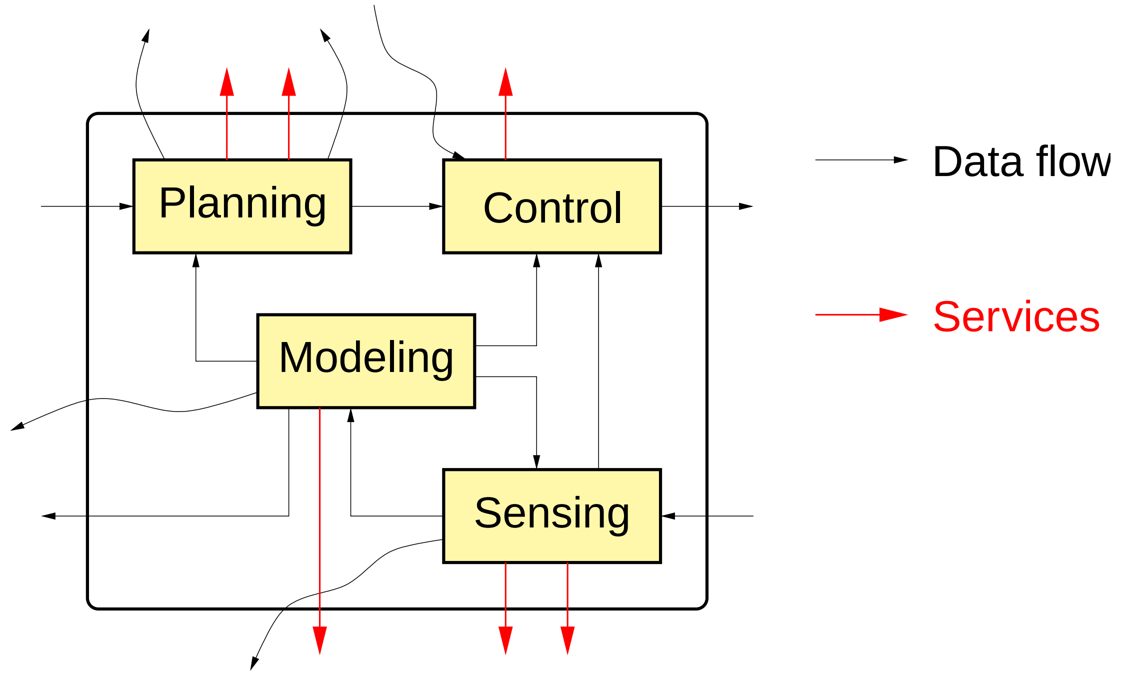
Motion control

- Motion control framework. All the yellow boxes could be implemented as components.



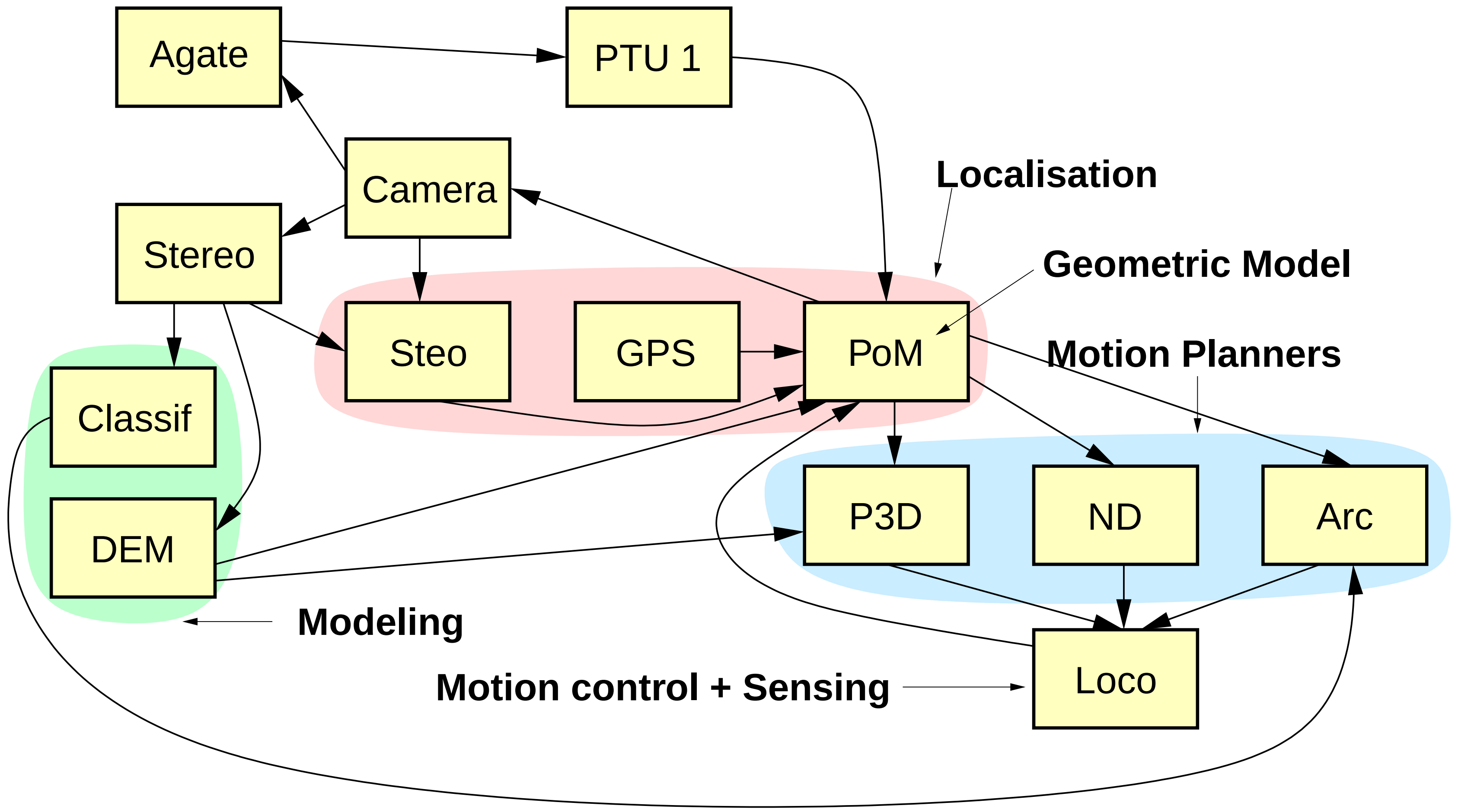
Examples — Motion control

- Motion control framework. All the yellow boxes could be implemented as components.



Exploration Task

→ implemented on an outdoor Marsokhod rover.



Purpose

- A formal description of a component will describe (and group together):

  $\longrightarrow$ the list of services,

  $\longrightarrow$ the services interface (input / ouput parameters),

  $\longrightarrow$ the set of codels for each services,

  $\longrightarrow$ the codels interface (imported / exported data, parameters, ...),

  $\longrightarrow$ as well as other information (not detailed here).

- The formal description is a text-based file (see the "tentative specification of components interface" document).

- The file is not used dynamically (at least not in its textual form). It is used during (at least) the link edition of the component, to select the set of modules (execution engine, codels).

Interface Description

- Tasks definition (time properties, ...).

```
thread <name> {
   priority: <number>;
   period:   <seconds>;
   stack:    <size>;

   start:    <function>;
   stop:     <function>;
}
```

Interface Description

- Services definition (inputs, outputs, exports, imports, codels, ...).

```
service <name> {
   doc: "short description of the service";

   thread: <name>;

   /* input/output parameters */
   input|output: ...;

   /* imported/exported data */
   import|export: ...;

   codel <name> {
      exec: <function>( [const] <variable>, ...);
      max-time: <seconds>;
      next: <codel> [, <codel>, ...];
   }

   ...
}
```

Discussion

$\longrightarrow$ Such a definition of components fills all requirements regarding *decoupling, modularity, reusability, configurability*, ...

$\longrightarrow$ The implementation is *free* (consequence of the aforementioned properties): CORBA, Sockets, C++, C, ...

Discussion

Discussion: Are you ok for...

- ... a formal description of components interface?
- ... codels?
- ... a decoupling between data and control flows?
- ... components that do not make decisions (control flow)?
- ... codels(?) that do not attempt to fetch data by themselves?
- ...