# A Component-Based Navigation-Guidance-Control Architecture for Mobile Robots

Nicolas Gobillot Charles Lesire David Doose

Onera - The French Aerospace Lab, Toulouse, France
firstname.lastname at onera.fr

**Abstract**

In this paper we will analyze a well known architecture in aerial robotics and generalize it to other mobile robots. We will then propose a normalized validation procedure for this generalized architecture pointing out some good practices when developing components.

## 1 Introduction

Robots are more and more autonomous, as they are involved in more and more complex environments, achieving more and more complex tasks. Anyway, however their autonomy skills are (from "simple" route following capabilities, to embedding complex AI planning algorithms), their autonomy always rely on a navigation architecture. This navigation architecture provides both localization information and locomotion capabilities.

In avionics, and therefore for aerial robots, this navigation architecture is used to be decomposed into *navigation*, *guidance* and *control* [1, 2]. For mobile (ground or marine) robots, this decomposition is not so common, but we can identify similar components on almost all existing architectures [3, 4].

In this paper, we propose to formalize robotics engineering traditions into a general pattern to design navigation, guidance, control (NGC) software architectures for mobile robots. This design pattern, along with some complementary best practices, is made to emphasize robustness and real-timeness of the architecture.

The NGC design pattern is a component-based generic software architecture. It then fits in Component-Based Software Engineering (CBSE), which is an essential design paradigm for robotic software development [5].

# 2 The NGC architecture

The main NGC pattern is based on three components:

- **Navigation:** this component uses the robot resources in order to locate itself in the world and to compute a coarse path to achieve the mission's objective.
- **Guidance:** the role of this component is to compute trajectories to reach the next waypoint provided by the navigation component; it includes reactive obstacle avoidance and local path planning.
- **Control:** its goal is to generate the commands applied to the robot's actuators in order to follow the guidance trajectories.

These three components are composed together to instantiate the NGC architecture (Fig. 1).

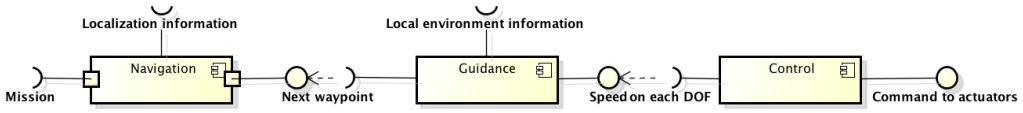


Figure 1: The NGC design pattern architecture

The navigation component has two inputs : the mission objective and some information on the robot localization. The former is represented as a waypoint to reach. The latter gives information about the robot localization in its environment. For instance, it may be a map from a SLAM algorithm along with an estimation of the robot position. The navigation component then outputs a local waypoint that represents a goal point to reach by the guidance component. The guidance component also takes into account local telemetry information in order to provide reactive obstacle avoidance while computing smooth trajectories. It then outputs the speed to apply to each DOF of the robot. Finally, the control component translates these speed commands into commands for the robot's actuators.

# 3 Component models

## 3.1 Mauve component models

In order to validate the component's general behaviour, we use the Mauve modelling framework [6] which brings a powerful scheduling analysis and an interface/codel separation within the analysis.

We split the component concept in two distinct parts : the interface with other components and the *codels*.

The former provides several means of interaction :

- **properties:** components can be configured by changing parameters in their properties;
- **services:** a component can let its peers to directly use one of its methods;
- **ports:** when a component needs to exchange data, it will use one of its input or output ports

The latter gives the component the methods and functions it needs.

The Mauve framework hardly relies on a formal definition of the modelled components. This definition is implemented using a domain specific language which is partially shown in the following components (listings 1, 2 and 3).

The Mauve model is made of several parts to fully describe a component:

- the codel declaration: it is done using the `fun` keyword followed by the codel name ant its typed parameters. The codel return type is also required;
- the component definition: this part is signaled by the word `component` followed by the component's name and contain several optional fields: such as `ports` or `attributes`. The former represents the component's ability to communicate with others. Every port has a specified type and is oriented as an input or an output. The `attributes` keyword will be used to declare internal variables to the component. On the other hand, the component model has a mandatory field which can be `update` or `statemachine`. The `update` keyword indicates where the component will loop at every cycle whereas the `statemachine` word will tell Mauve that the default state machine of the component will be redefined.

## 3.2 Codels

The word codel stands for Elementary Code. This code provides the functionalities and the algorithms to the host component. These codels are meant to be target-independent and should be written in any analyzable language. It also allows its functionalities to be shareable and easily maintainable. In

the general case it is a good practice to write reentrant and thread-safe code without any unbounded loop.

## 3.3 NGC components

We will now put more details on what is really in the Navigation, Guidance and Control Mauve models. First, the navigation component is made of the default state machine (listing 1).

Listing 1: Mauve model of the Navigation component

```
fun update_plan(position: Pose, mission: Pose): Plan
fun compute_next_waypoint(position: Pose, plan: Plan)): Pose

component Navigation {
  ports {
    input currentPosePort: Pose
    input missionPort: Pose
    output nextPort: Pose
  }
  attributes {
    mission: Pose
    plan: PoseList
    position: Pose
    nextPos: Pose
  }
  update {
    if (missionPort.hasNewData) then {
      mission = read missionPort;
      plan = update_plan(position, mission);
    }
    if (currentPosePort.hasNewData) then {
      position = read currentPosePort;
      nextPos = compute_next_waypoint(position, plan);
      write nextPos in nextPos_port
    }
  }
}
```

The Navigation component reacts to the `currentPosePort` and `missionPort` input ports which respectively gets the local position of the robot and a mission objective (in this example, the mission is a position on a map). If a new mission has been given, it will call the `update_plan` codel to build a new plan. On the other hand, when the localization of the robot has changed, it will check if the output waypoint has been reached and if it needs to be changed using the `compute_next_waypoint` codel. Lastly it will output a waypoint by the `nextPort` output port.

The second component in the architecture, the Guidance component, has a user-defined statemachine (listing 2) to bring the necessary behaviour.

Listing 2: Mauve model of the Guidance component

```
fun stop_robot(): Speed
fun nextPoint(currentPose: Pose, targetPose: Pose, localMap: Scan): Pose
fun guidance(nextPose: Pose, currentPose: Pose): Speed

component Guidance {
  ports {
    input targetPosePort: Pose
    input localMapPort: Scan
    output speedPort: Speed
  }
  attributes {
    speedPtr: Speed
    currentPose: Pose
    targetPose: Pose
    localMap: Scan
    nextPose: Pose
  }
  statemachine {
    initial state Arrived {
      run {
        if (isArrived(currentPose, targetPose)) then {
          speedPtr = stop_robot()
        }
      }
      handle {
        write speedPtr in speedPort
      }
      transition toMoving -> Moving
    }
    state Moving {
      run {
        if (targetPosePort.hasNewData) then {
          targetPose = read targetPosePort
        }
        if (localMapPort.hasNewData) then {
          localMap = read localMapPort;
          nextPose = nextPoint(currentPose, targetPose, localMap);
          speedPtr = guidance(nextPose, currentPose);
          }
        }
      }
      handle {
        write speedPtr in speedPort
      }
      transition toArrived [isArrived(currentPose, targetPose)] ->
          Arrived
    }
  }
}
```

---

Every state of the state machines have five parts:

- `entry`: this part is executed when the state machine arrives in a new state;
- `run`: this part is executed after the `entry` part has finished its execution;

- `transition`: after the execution of the `run` part, if a transition is validated, this step will call the `exit` part and the next state `entry` is executed;
- `handle`: this step will be executed after `run` if no transition has occurred;
- `exit`: this part is executed after a transition.

The Guidance custom statemachine has two states:

- the `Arrived` state which is a state when the robot is just stopped by the `stop_robot` codel. The `initial` keyword indicates that when the component will be started for the first time, the state machine will begin in this state;
- the `Moving` state: it will check if a new local map has arrived on the `localMapPort` input port and call the `nextPoint` codel in order to compute a local path avoiding potential obstacles. It will also check if a new waypoint has been given by the Navigation component.

In both cases, the Guidance outputs the target's DOF speeds to the next component.

And the last component of the base architecture is the Control (listing 3). Like the Navigation model it uses the default statemachine.

LISTING 3: Mauve model of the Control component

```
1  fun TTRKcontrol(speedVector: Speed, actuatorCommand: Actuator): int
2
3  component Control {
4    ports {
5      input speedPort: Speed
6      output actuatorCommandPort: Actuator
7    }
8    attributes {
9      speedVector: Speed
10     actuatorCommand: Actuator
11   }
12   update {
13     if (speedPort.hasNewData) then {
14       speedVector = read speedPort;
15       TTRKcontrol(speedVector, actuatorCommand);
16       write actuatorCommand in actuatorCommandPort
17     }
18   }
19 }
```

This component reacts to its `speedPort` input port to compute the orders given to the target actuators using the `TTRKcontrol` function.

This design pattern has several good points:

- it is made of numerous small components that are easy to maintain upon time;
- is it generic, and can then adapt to several robot platforms (aerial robots, ground vehicles, boats or submarines) by only adapting data types and codels (i.e. algorithms) to the targeted platform;
- each component is meant to have a real-time behaviour and keep a relative reactiveness depending on their hierarchical level.

To build a coherent and robust NGC architecture, we are proposing some complementary best practices:

- as soon as it is possible, create unified and normalized small data structures;
- use a limited number of functions on each component, as it improves modularity and maintainability of the architecture;
- regarding execution, create independent thread for each component, and use periodic activities instead of causal links: it provides more robustness to the architecture as a component can continue to work even if another component has failed; it is then easier to include recovery or safety strategies in the architecture.

## 3.4 Examples

The whole NGC architecture has been implemented on two targets: the first one is a TTRK robot with an ARM powered Gumstix Water[1] running a real-time Linux kernel. The second is a simulated Pioneer-3DX on a x86 computer running the MORSE simulator [7]. The only things that changed between the two implementations were the Control's Codel and the adapters to and from target-specific data types. In both cases, all the components were deployed using the Orocos middleware [8] and logged by the LTTng trace toolkit[2].

[1] `https://www.gumstix.com/store/product_info.php?products_id=228`

[2] `https://lttng.org`

The software architecture is based on the main NGC pattern which was completed by adapters to convert some of the non-generic data types, a joystick and a switch to keep manual control on the system if desired. A planner is also plugged onto the Navigation component to provide potential parallel planning with different algorithms (Fig. 2).

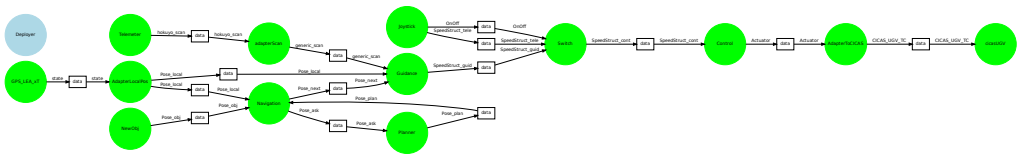

Figure 2: Architecture embedded on the TTRK and Pioneer robots

# 4 Real-Time analysis

The previously described component models are meant to be run on Real-Time operating systems, even if it will still work fine on non Real-Time systems.

In order to ensure a good and safe behaviour of the component-based robotic system, we need to validate the real-time constraints of the embedded components. Therefore we have defined an analysis protocol for that purpose.

## 4.1 Methodology

The very first step of the analysis is the formal validation of the component models by verifying for example the non-locking aspect of the task. Once the model has been checked, an instance of the component can be generated using the code generation feature of the Mauve framework in order to ensure the validity of the running code.

All the Codels must be developed keeping in mind that WCETs have to be computed. In this paper we will be using the C language in order to use tools such as Frama-C[3] and OTAWA [9] for respectively validate the properties of the algorithms and compute their Worst-Case Execution Time (WCET).

The last step in this procedure is to deploy the whole set of components on the target in order to log its execution and compare the theoretical results with the real ones.

[3] `http://frama-c.com`

## 4.2 Tracing example

The LTTng tracing toolchain gives us the real execution times of each component. We will for example focus on the Control component (Fig. 3). This plot represents the probability to get execution times for the Control component running on the TTRK's Gumstix. It shows three main peaks which can be set in two groups:

- the first one on the left represents the state when the component does not receive any new data; it therefore has nothing new to compute and returns rapidly;
- the second group is made of the middle and right peaks on the plot and corresponds to the case when the codel is called; these two peaks are two branching possibilities within the codel.

This kind of information is interesting because it will tell us if the worst case execution time will be reached frequently or not and potentially optimize parts of the component or codels.

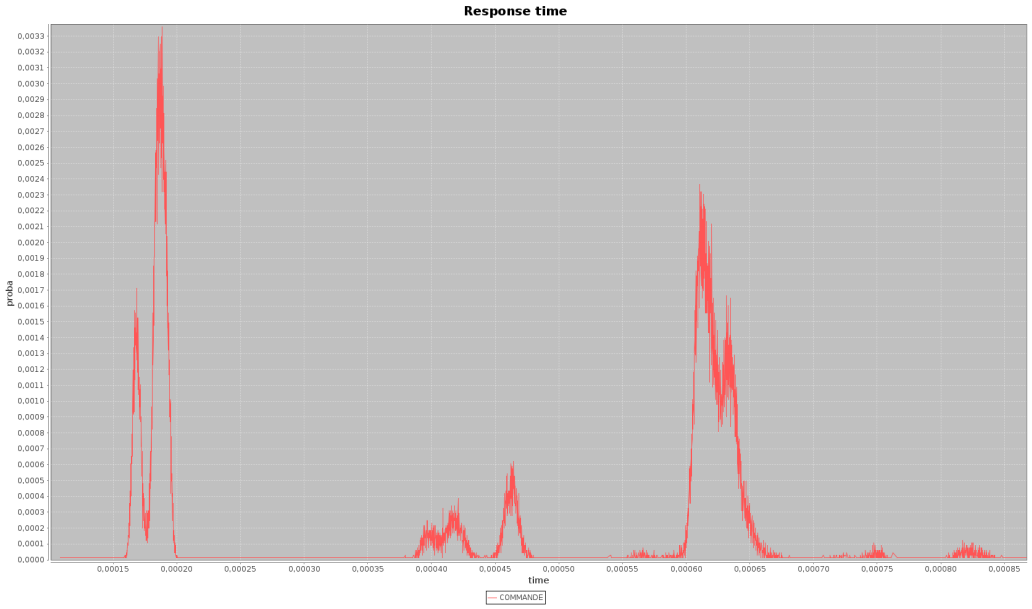


Figure 3: Distribution of the execution time of the Control component

The deployment procedure is set on three parts:

- the first part is the importation of the component models previously described using the `import` command;

- the second step is the instantiation of the needed components by using the keyword `instance` followed by an instance name and the name of the component. When instantiating the components, we also setup the connections between the input and output ports;
- the last step contains the properties of the components and is signalled by the `deployment` command. Within this part we specify the codels best and worst case execution times and the instantiated components activities. The component activities are set using the `activity` keyword and contains the components priority, its period and its deadline.

An extract of the deployment pattern is shown on the listing 4.

Listing 4: Deployment of the NGC components in Morse

```
1  import "Navigation.xma"
2  import "Guidance.xma"
3  import "Control.xma"
4
5  // Architecture
6  instance Navigation: Navigation {}
7  instance Guidance: Guidance {
8    port targetPosePort data Navigation.nextPort
9  }
10 instance Control: Control {
11   port speedPort data Guidance.speedPort
12 }
13 ...
14
15 // Deployment
16 deployment {
17   // Navigation
18   Navigation_codel = 261..488
19   activity Navigation {
20     priority = 7
21     period   = 50000
22     deadline = 50000
23   }
24   // Guidance
25   Guidance_codel = 129..1148
26   activity Guidance {
27     priority = 4
28     period   = 10000
29     deadline = 10000
30   }
31   // Control
32   Control_codel = 109..1818
33   activity Control {
34     priority = 2
35     period   = 5000
36     deadline = 5000
37   }
38   ...
39 }
```

The deployment in Mauve needs times expressed in integers, in the above example the times are written in microseconds. We have to provide the best

and worst case execution times for every codel of every component deployed. The periods and deadlines of the components are required as well as their priority: the lower the value, the higher the priority.

We can then simulate the deployment in order to compute timelines of the scheduled tasks (Fig. 4). On the diagram, the tasks are ordered in decreasing priority and we can clearly see that there is not much time left for the high level tasks such as the Planner. In this particular example, the planner has nothing to compute and therefore the task set is schedulable.

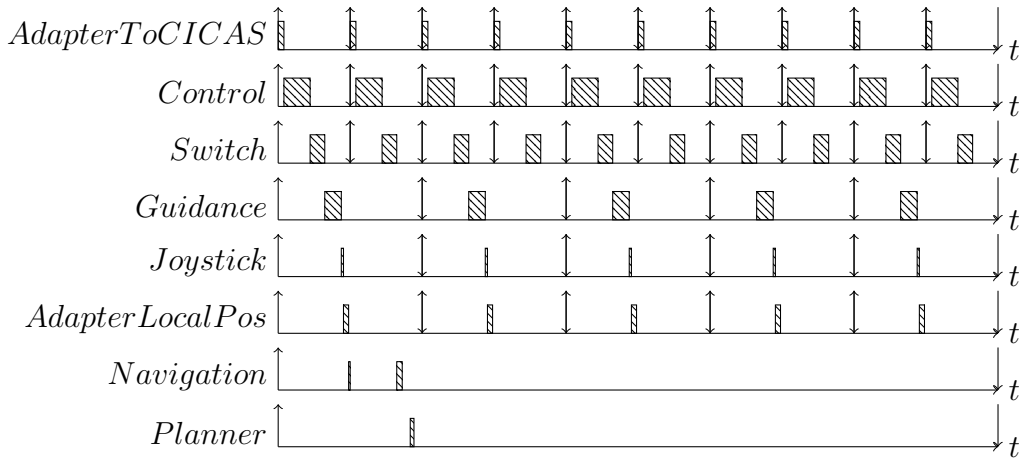


Figure 4: Task scheduled by a Rate-Monotonic algorithm

# 5 Conclusion

In this paper, we have presented the Navigation-Guidance-Control design pattern.

It formalizes a well known architecture in aerial robotics to mobile robots in general.

This design pattern brings to light best practices for embedded software engineering in robotics.

In the future work, we plan to implement this architecture on different robots, not only the tracked TTRK or the Pioneer-3DX but also on an aerial robot such as a quadrotor. It will allow us to check the multi-target feature the NGC design pattern brings.

Another work in progress is the validation of behavioural properties such as the fault tolerance when a component or a sensor fails.

# References


[1] F. Kendoul, "Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems," *Journal of Field Robotics*, vol. 29, no. 2, 2012.

[2] J.-H. Kim, S. Sukkarieh, and S. Wishart, "Real-Time Navigation, Guidance, and Control of a UAV Using Low-Cost Sensors," in *Field and Service Robotics (FSR 2003)*, (Lake Yamanaka, Japan), 2003.

[3] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila, "Autonomous Rover Navigation on Unknown Terrains," *International Journal on Robotic Research*, vol. 21, no. 10-11, 2002.

[4] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "A deliberative architecture for AUV control," in *International Conference on Robotics and Automation (ICRA 2008)*, (Pasadena, CA, USA), 2008.

[5] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II)," *IEEE Robotics and Automation Magazine*, vol. 17, no. 1, 2010.

[6] C. Lesire, D. Doose, and H. Cassé, "MAUVE: a Component-based Modeling Framework for Real-time Analysis of Robotic Applications," in *Workshop on Software Development and Integration in Robotics (SDIR 2012)*, (Saint-Paul, MN, USA), 2012.

[7] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, and S. Stinckwich, "Simulating complex robotic scenarios with morse," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, (Tsukuba, Japan), 2012.

[8] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *International Conference on Robotics and Automation (ICRA)*, (Barcelona, Spain), 2005.

[9] C. Rochange and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pp. 35–46, 2010.