Journal of Software Engineering for Robotics 1(1), September 2009, 123-126
ISSN: 2035-3928

MAUVE Runtime: a component-based middleware to reconfigure software architectures in real-time

David DOOSE[1] Christophe GRAND[1] Charles LESIRE[1]
[1] ONERA – The French Aerospace Lab, Toulouse, France
firstname.lastname@onera.fr

Abstract—Developing robotic applications requires to design and implement complex software architectures. These architectures must embed advanced algorithms that include capacities to adapt to unforeseen events like external disturbances, sensor or actuator failures. To improve the system robustness, its behavior should be adapted at runtime by a reconfiguration of its software architecture. Such reconfiguration must be done safely and efficiently, while ensuring functional constraints and a minimal quality of service of the system. Among these constraints, preserving real-time properties of the reconfiguration process is a key feature. In this paper, we present the design of a new component-based middleware that allows to perform software architecture reconfigurations with a focus on real-time constraints.

Index Terms—Control architectures and programming, Reconfigurable computing, Real-time systems, Middleware, Robotics.

1 INTRODUCTION

ENSURING software dependability of complex robotic applications is an essential issue in the democratization of autonomous robots usage in everyday life. Many examples enlighten the critical need for safety in autonomous systems, among which drone applications, self-driving cars, or robot-human cooperation.

All these applications need for *autonomous* and *safe* robots: the robotic system must be able to adapt all along its mission (adapt to environment changes or to system failures). But the robotic system has also to be reliable, in order to be accepted both by some regulation office, and by end-users.

To address the problem of developing the software part of such autonomous and safe robots, roboticists have largely adopted the use of middlewares, that ease the development process, regarding collaborative development, separation of roles, maintainability and reusability of the software pieces.

Nevertheless, the robotic middlewares have almost not addressed two key issues of safe and autonomous robot development: (1) reconfiguration mechanisms, and (2) real-time guarantees.

Regular paper – *Manuscript received April 19, 2009; revised July 11, 2009.*

- *Corresponding author: Charles Lesire, ONERA/DTIS, 2 avenue Edouard Belin, 31400 Toulouse, France,* charles.lesire@onera.fr

In this paper, we present MAUVE Runtime, a new middleware for developing component-based software architectures with a focus on real-time guarantees and reconfiguration mechanisms. None of the existing middlewares used in robotics provides *real-time safe* reconfiguration mechanisms.

The paper is organized as follows. Next section analyses the available middlewares or development processes, with respect to real-time concerns and reconfiguration mechanisms. Section 3 describes the concepts of the MAUVE Runtime. Some benchmarking regarding the real-time implementation is given in Sec. 4. Finally, a robotic application is described in Sec. 5.

2 RELATED WORKS

Modern approaches in software development of robotic applications settle on the paradigm of Component-Based Software Engineering (CBSE, [1]). A component-based design allows the software architect to build a robotic architecture by assembling existing software components [2]. These software components are made of two parts: their communication interface and their internal behaviour. Communications are driven by connecting ports or by service calls between two or more components. The component's behaviour is often defined by a state-machine, that allows to define several operational modes, including some degraded mode to be robust to sensor or software failures.

This component-based approach is often associated with modelling languages. These modelling languages are almost every time Domain Specific Languages (DSLs): they are

dedicated to model component-based software architecture, including specificities related to an application or some characteristics of the architecture in development.

2.1 Real-time middlewares

Some of the DSL-based processes have dealt with real-time features. For instance, BIP [3] allows to model components based on a *Behaviour-Interaction-Priorities* structure. From a BIP model, it is possible to generate code that will be executed by a real-time BIP engine. Genom [4] allows to model components, assigning several tasks to components, each task supporting the execution of services. Genom models can be mapped to several middlewares/systems, including Pocolibs (the original Genom middleware), that focuses on data sharing between processes, and can run on VxWorks. SmartSoft/SmartMARS [5], [6] describes components and how they are mapped to real-time tasks of the system. Some schedulability analysis is made, as far as the execution time of the components are available. In MAUVE [7], components are mapped to only one task, and a periodic state-machine is used to model the component behavior in order to improve the accuracy of schedulability analyses [8]. MAUVE generates ROS/Orocos integrated architectures.

Some of these processes use standard middlewares to implement components and architectures, either general purpose middlewares (like DDS in Smarsoft [5]), or robotic middlewares. For instance, Genom [4] generates Pocolibs, ROS [9], or Orocos [10] code. MAUVE generates ROS/Orocos integrated code.

However, when developing software for robotic application, the use of the API of the middleware is still very common. Most of the robot developers are programmers. Even if the model-based development is clearly the way to go, robot architecture should be able to integrate both generated components and hand-written components. Having a design process promoting model-based engineering and compliant with a usable middleware is clearly a must. ROS [9], while being the most widespread middleware, does not provide any real-time specific API, nor any real-time execution model of the nodes. The reference middleware regarding real-time features is Orocos [10]. It can be directly used by programmers, through the ROS/Orocos integration pipeline. Rock [11], Genom [4], MAUVE [7] and HyperFlex [12] propose a component-based model mapped to Orocos components.

As far as we know, MAUVE is the only component-based process that has used model of the component behaviours to evaluate accurate schedulability analysis of the software. However, Orocos suffers some drawbacks, among which the lack of determinism in the deployment process [13], and the fact that it is almost impossible to exhibit a model from hand-coded component as the Orocos API is too far from the MAUVE execution model.

2.2 Reconfigurable middlewares

Reconfiguration is another key issue when implementing adaptable functions. Few works related to robotic software development have considered the capability to reconfigure software architectures. While stopping a component and starting another one is always possible, the impact of real-time characteristics of the software architecture is always unknown. In [14], the authors propose to address it by implementing domain-specific real-time operating system mechanisms together with port-based object software abstraction. It was developed for robotic manipulator to realize dynamic reconfiguration of controller. In [15], the authors introduce a software framework based on CORBA that allows to perform reconfiguration of distributed architecture based on components replacement while minimizing disruption to the application and limiting run-time overhead to meet real-time requirements of critical robotic applications. In [16], they propose an adaptive middleware system for distributed sensors, but only to reduce the power consumption by dynamically trading off performance requirements where real-time considerations are considered as a problem of collected data synchronization. In [17], authors consider the need for dynamic self-adaption in modules of systems such as robots, in order to provide fault-tolerance and propose the *Containment Unit Architecture* that supports the switching between different resource configurations and operational components at run-time as long as the transition between configurations is relatively simple. Orocos and Smartsoft have also tried to provide guarantees when rewiring the architecture, i.e. when disconnecting and reconnecting data ports.

3 MAUVE Runtime

The rationale of developing the MAUVE Runtime comes from the need to provide reconfiguration mechanisms as well as a real-time execution model at the middleware level. It is indeed essential to have a middleware implementation that matches the formal behaviour and execution model of the software architecture in order to be able to make accurate analysis.

3.1 Overview

The MAUVE Runtime is a component-based middleware that allows to perform software architecture reconfigurations with a focus on real-time constraints. The software architecture is described as components connected to each others through resources, and configured through properties. Components and resources are defined by a Shell, defining the external part of the component, and a Core, defining the inner data and functions of the component. Moreover, components have a Finite State-Machine, defining the temporal behavior of the component ; resources have an Interface, defining the services provided by the resource. An Architecture is then built by instantiating, connecting and configuring components and

resources. Every element in MAUVE Runtime is *Configurable*, meaning that the element has an internal state that limits when the inner data of the element are modifiable.

3.2 Configurable elements

In the MAUVE Runtime every element is configurable: the shells, the cores, the finite state machines, the interfaces, the components, the resources and the architectures. Once an element is configured, its characteristics cannot be modified anymore. For instance, properties of shells cannot be modified, clocks of state-machines cannot be modified, or components of an architecture cannot be modified. Architecture reconfiguration will then need to unconfigure elements first, then reconfigure them (see 3.9).

3.2.1 Configurable state machine

Each configurable element can be in (at least) two states: *not configured* or *configured* (Fig. 1). At first, the element is *not configured*. Calling the method *configure()* can change its state into *configured* in case of success. Calling the method *cleanup()* changes its state into *not configured*.

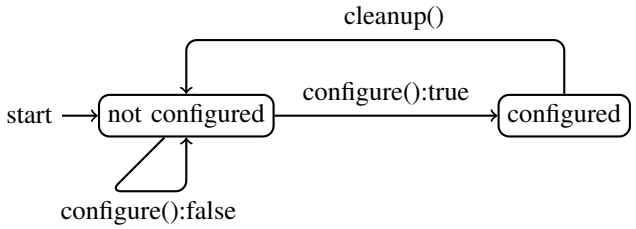

Fig. 1: Configurable state machine

3.2.2 Configurable dependency

The different elements of the MAUVE Runtime are structured by composition relations and dependencies. For example a component contains a shell, a core and a finite state machine. Each core depends on its shell and each finite state machine depends on its core (and by transitivity depends on its shell).

The composition relations and dependencies between elements produce a complete order. For the components, this order is:

$$(Shell \leftarrow Core \leftarrow FSM) \leftarrow Component \qquad (1)$$

Consequently, while configuring a component, its shell is configured first, then its core, then its finite state machine and finally the component is configured. Moreover, if (for example) the configuration of the core fails, then neither the FSM nor the component can be configured.

The *configurable* topological order is important for the architecture configuration but also for the reconfiguration process. The reconfiguration of a component core indeed implies the cleanup of its state machine, but there is no impact on its shell. This mechanism is important because it allows to reconfigure inner elements without any impact on the rest of the architecture.

3.2.3 Configurable hooks

By default the *configure* method simply try to configure the inner elements and returns true, and the *cleanup* simply cleans the inner elements. The behavior of the *configure* and the *cleanup* can be extended by respectively overriding the *configure_hook* and *cleanup_hook* methods in the corresponding element.

3.3 Common elements

Common elements to components and resources are presented in this section.

3.3.1 Property

Each shell, core, finite state machine and interface can contain several properties. The properties are used to configure the corresponding element. The value of the properties can only be changed when the element is *not configured*. Once in the *configured* state, its properties can only be read.

3.3.2 Shell

The shell describes the external part of a component or a resource. It contains ports and properties. The shell is the only visible part outside the component/resource. Ports are used to exchange data between components and resources. Ports are divided in four types depending on their capabilities:

- `ReadPort<T>` can *read* data of type `T`;
- `WritePort<T>` can *write* data of type `T`;
- `CallPort<R, P...>` can *call* a method with `P...` parameters and retrieve a result of type `R`;
- `EventPort` is used to trigger an action to the connected service.

Listing 1 shows the definition of the shell `ObstacleAvoidance` (further described in Sec. 5). It contains three read ports and two write ports.

Listing 1: Shell `ObstacleAvoidance`

```
1  struct ObstacleAvoidanceShell : public Shell {
2    ReadPort<Pose2D > & pose =
3      mk_read_port<Pose2D >("pose", Pose2D());
4    ReadPort<Point2D> & goal =
5      mk_read_port<Point2D >("goal", Point2D());
6    WritePort<Velocity> & command =
7      mk_write_port<Velocity >("command");
8    WritePort<double> & distance_to_goal =
9      mk_write_port<double >("distance_to_goal");
10   ReadPort<LaserScan> & scan =
11     mk_read_port<LaserScan >("scan", LaserScan());
12 };
```

3.3.3 Core

The core describes both the algorithmic part and the inner data of the component/resource. Thus, the core defines attributes and methods. The core depends on the shell which means its methods can interact with the shell. Consequently a core can read properties, and read and write data, on ports.

Listing 2 defines the core of the `PotentialField`. Its internal variables are the current goal and the current robot position, as well as the path that is being executed. It defines a set of methods that will be called from the state-machine.

Listing 2: Core `PotentialField`

```
struct PotentialFieldCore
  : public runtime::Core<ObstacleAvoidanceShell> {

  Property<double> & obstacle_distance =
    mk_property("obstacle_distance", 0.8);
  Property<double> & safety_distance =
    mk_property("safety_distance", 0.8);
  Property<double> & nu = mk_property("nu", 0.2);
  Property<double> & psi = mk_property("psi", 0.2);

  bool new_goal();
  bool is_arrived();
  ...
private:
  ...
};
```

3.4 Components

As previously mentioned a component is made of a shell, a core and a finite state machine (FSM). The core depends on the shell and the FSM depends on the core. Thus the core can use shell elements, and the FSM can use core and shell elements.

Finite State-Machine model in Orocos [10] allows to define only Periodic State-Machine, i.e. state-machines that execute one and only one state at each period, with a fixed period. Genom [4] allows to specify either successive transitions during a same period, or transitions that must yield on the next component period. In MAUVE, we have extended this behaviour by defining clock-based FSMs, composed of two different type of states, and transitions:

- *execution* states are aimed to execute code (i.e. methods defined in the component's core);
- *synchronization* states *pause* the component until its internal clock reaches a specific value;
- *transitions* between states are ordered and guarded.

This model allows to define sequences of execution states, then executed without waiting, and synchronization states with different values, then leading to managing different "periods" of execution within the same component.

Listing 3 defines the `PotentialFieldFSM` of the control component. It contains three execution states and three synchronization states.

Listing 3: FSM of the `PotentialField` control component

```
struct PotentialFieldFSM
  : public FiniteStateMachine
  <ObstacleAvoidanceShell, PotentialFieldCore> {

  ExecState<PotentialFieldCore> & wait_goal =
    mk_execution("wait_goal",
                 &PotentialFieldCore::read_input_goal);
  SynchroState<PotentialFieldCore> & sync_wait =
    mk_synchronization("sync_wait", ms_to_ns(100));
  ...
  bool configure_hook() override {
    set_initial(wait_goal);
    mk_transition(wait_goal,
                  &PotentialFieldCore::new_goal,
                  reach_goal);
    set_next(wait_goal, sync_wait);
    ...
    return true;
  };
};
```

Figure 2 represents the corresponding clock-based FSM. The execution state WG is where the arrival of a new goal is checked. If no goal has arrived, the component waits $100ms$ in the synchronization state SW. Otherwise the component sends commands to reach the goal, in the execution state RG. If the goal is reached then the execution state AG is executed, and the component waits $50ms$ before checking for a new goal. Reaching the goal is implemented as a control loop with a period of $10ms$ (state SR).

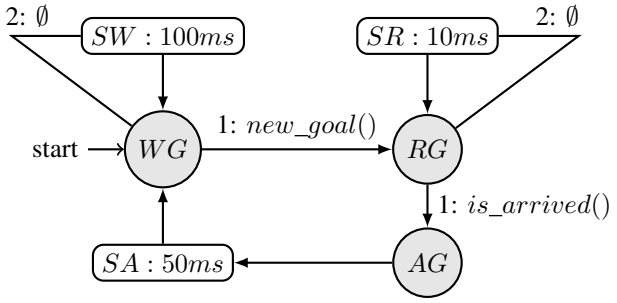

Fig. 2: Clock-based FSM of the control component: circles represent execution states, W_i are wait states.

Figure 3 illustrates the timeline of the control component for path: $\langle WG, SW, WG, RG, SR, RG, SR, RG, AG, SA, WG\rangle$.

Fig. 3: Control component timeline.

3.5 Resources

The MAUVE Runtime has a generalized inter-components communication scheme with the notion of *Resource*. A *Re-*

source has two main objectives: first it contains data, second it provides several services to interact with its inner data. Component ports are connected to a resource using its corresponding services.

A *Resource* is made of a shell, a core and an interface. The core depends on the shell and the interface depends on the core. Thus the core can use shell elements, and the interface can use core and shell elements.

An interface is a configurable element and defines services, it depends on a core (and a shell). Like ports, services are divided in four types depending on their capabilities: `ReadService<T>`, `WriteService<T>`, `CallService<R, P...>` and `EventService`. A method of the corresponding core is associated with each service of the interface.

Listing 4 defines the interface of the *ROS Publisher* resource. The main purpose of the *ROS Publisher* resource is to ease the communication from the MAUVE Runtime to ROS. The interface has a single service *write* which converts a data of type `T` to a ROS type `ROS_T` and publishes it to the previously defined topic. The ROS topic is specified in the shell of the resource and the conversion method is defined in the core.

Listing 4: *ROSPublisher* Interface

```
template <typename T, typename ROS_T>
struct PublisherInterface:
  public Interface<
    PublisherShell<T, ROS_T>,
    PublisherCore<T, ROS_T>> {
  WriteService<T> & write =
    mk_write_service<T>("write",
      &PublisherCore<T, ROS_T>::publish);
};
```

The basic resources `SharedData` and `RingBuffer` are provided by the MAUVE Runtime to mimic classical data flow communication, and are accessible through read and write services. A specific resource can be created by defining its shell, its core and its interface. Moreover, defining specific resources with advanced processings allows to reduce data communications. For example, sharing the map in a resource between a SLAM component and a Navigation component reduces the number of copies of the map and the number of data sent (less memory used, lower computation time).

It is important to notice that resources do not have any internal activity, which means that their core methods are executed when a component interacts with the resource.

From a real-time point of view, access to shared data may be a real issue because it can induce priority inversion, deadlock, etc. In order to avoid those problems and to maintain the determinism of the system, *Resources* are the only way to exchange data between components. Moreover, *Resource* access has been secured to provide a safe inter-components communication, thanks to three points:

- each *Resource* is protected by a mutex to ensure data integrity;
- due to the implementation of the services and the protection of the *Resource*, the interlining access of resources is not possible[1];
- the *Resource* mutex is configured to use the real-time `PIP` protocol [18] to guarantee determinism and avoid priority inversion.

3.6 Architecture

Components and resources are instantiated within the architecture. Figure 5 illustrates the architecture configuration in which the properties of the different elements are specified and the components and resources ports are connected to the services of the resources. Finally, the components and the resources are configured.

Listing 5: architecture

```
struct NavigationArchitecture : public Architecture {
  // Components
  Hokuyo & hokuyo = mk_component<Hokuyo>("hokuyo");
  PotentialField & control =
    mk_component<PotentialField>("control");
  ...
  // Resources
  SharedData<LaserScan> & scan =
    mk_resource< SharedData<LaserScan> >("scan",
        LaserScan());
  ...
  bool configure_hook() {
    // Connections
    hokuyo.shell().scan.connect(scan.interface().write);
    control.shell().scan.connect(scan.interface().
        read_value);
    ...
    // Properties
    hokuyo.shell().device = "/dev/hokuyo";
    hokuyo.fsm().period = ms_to_ns(25);
    ...
    // Configure
    return Architecture::configure_hook();
  };
};
```

3.7 Deployment

Components are then mapped to real-time tasks by the *Deployer*. The MAUVE Runtime deployer has been designed to have a complete control on the synchronization between tasks.

Timeline of Fig. 4 illustrates the deployment of two tasks (τ_1 and τ_2) executed on the same processor; the task τ_1 has an higher priority than τ_2.

At the beginning of the execution of the system, the deployer is executed (between t_0 and t_1). Its main purpose is to get and maintain a unique clock reference[2] for all the tasks, and then launch the real-time tasks. Tasks τ_1 and τ_2 are synchronized by the deployer; at t_1 they are released (at the

1. this point prevents deadlocks
2. the system start time is t_0

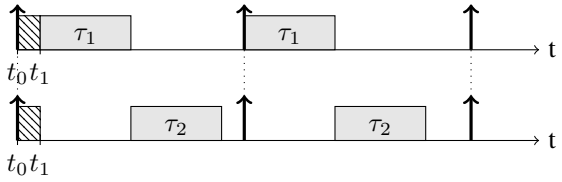

Fig. 4: Tasks synchronization

Listing 6: Reconfiguration code snapshot: the core of component `control` is replaced.

```
1 time_ns_t clock = control_task->get_time();
2 control_task->stop();
3 control->cleanup_core();
4 control->replace_core<FiniteTimeAvoidanceCore>();
5 control->configure_core();
6 control_task->start(clock, nullptr);
```

exact same time) and compete to the processor access. The task τ_1 is executed before τ_2 because it has a higher priority. Since the tasks, τ_1 and τ_2 have the same period and the same clock reference, their synchronization is maintained during all the execution.

3.8 Implementation

The MAUVE Runtime implementation relies on C++11 and highly use templates. Object oriented programming principles and subtyping allows to define subtypes of the different elements of the MAUVE Runtime such as Shells, Cores, ... The subtypes can be used in the architecture specification or in the real-time reconfiguration.

The MAUVE Runtime aims to be Real-Time. To achieve this objective the MAUVE Runtime deployment and execution is based on the Real-Time POSIX API. In practice each component is mapped into a thread; the real-time scheduler `SCHED_FIFO` is used to schedule the threads. The services of the resources use *mutex* with the real-time protocol `PIP` [18] to guarantee determinism and avoid priority inversion.

The MAUVE Runtime currently runs on different hardware (various ARM and x86 processors) and on different Linux distributions.

The MAUVE Runtime is open-source, under the LGPL license. The source code is available at https://gitlab.com/mauve/mauve_runtime. Description and documentation of MAUVE are available at https://mauve.gitlab.io.

3.9 Reconfiguration mechanisms

The MAUVE Runtime allows to perform reconfigurations by relying both on the component model and on the real-time deployment. It is indeed possible to stop a component, reconfigure it by changing its properties, replacing its shell, its core, or its FSM, and restart it again while maintaining synchronization with the other components. Listing 6 shows a code snapshot performing a real-time reconfiguration of the control component.

In order to make a sound reconfiguration, with respect to real-time task synchronization, we first get the reference clock of the task of the component to be reconfigured. Then we stop the task, and replace the core of the *control* component. We then configure this core and restart the task according to the same clock reference. In this example, a supervision component initiates the reconfiguration of the control component by switching its `PotentialFieldCore` to the `FiniteTimeAvoidanceCore`. This reconfiguration is further detailed in Sec. 5. Figure 5 illustrates the real-time behavior of such a reconfiguration: the second task is stopped in order to replace its core C_A by the core C_B (during R). The task is restarted, the synchronization being maintained.

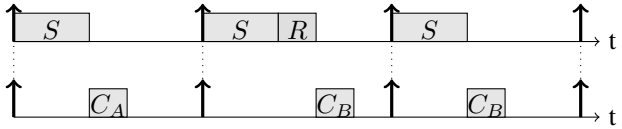

Fig. 5: Task synchronization during reconfiguration of a core C_A into C_B by a supervision component S.

While considering real-time critical systems, it is important to maintain a safe behavior even during the reconfiguration process. Consequently, the inner state and values of the components have to be correct, which means that the values of the attributes are conform to the component core algorithmic. To achieve this objective, one prerequisite is to never stop/kill a component during its computation (running an execution state). That is why the reconfiguration can only be triggered between two states of the component finite state machine.

4 Benchmark

In this section, we present an evaluation of the MAUVE Runtime implementation on a simple benchmark. The source code of this benchmark is available at https://gitlab.com/MAUVE/benchmarking. The traces that allowed to make the following tables are available at https://mauve.gitlab.io/files/benchmark_runtime_joser.tar.gz.

4.1 Benchmarking process

This benchmark is done with a simple architecture made of two components, A and B. A disturbing process is also launched to analyze its impact on the scheduling on the two main components. The computation time of this process is between 10% and 50% of the CPU.

Each component is a simple periodic component, with no input/output. It has a time-consuming function (implemented using a loop on the `getrusage` instruction to evaluate the CPU time associated to its thread).

This architecture has been implemented using several middlewares and configurations.

- ROS: each component is implemented as a simple node containing a `while` loop with a ROS rate to manage task periodicity; these nodes are non real-time, as ROS does not provide an API to build real-time tasks;
- OROCOS: each component is implemented as a real-time periodic component; the deployment has been implemented in C++ in order to configure the behavior of the `deployer`;
- MAUVE: each component is implemented as a real-time component with a simple periodic state-machine.

The configuration of the tasks involved in the benchmark are presented on Table 1.

task	A			B		
	P	T	C	P	T	C
ROS	0	0.1	0.02	0	0.1	0.01
OROCOS	20			30		
MAUVE						

TABLE 1: Configuration of the benchmark tasks. P: priority. T: period. C: computation time. The disturbing process has a period of 10ms, and uses a CPU time randomly drawn at each iteration between 1 and 5ms.

In every run, task A is launched first, then component B, in order to evaluate the real-time behavior of the runtime regarding priority management.

We moreover implemented three variants of the OROCOS deployment, where we changed the characteristics of the `deployer` component, that is in charge of configuring and starting the components:

- in the `-prio` configuration, the deployer has a higher priority than the components;
- in the `-bg` configuration, the deployer has a lower priority than the components;
- in the `-isolated` configuration, the deployer runs on a specific CPU core.

4.2 Evaluation of real-time execution

Table 2 shows statistics gathered on executing the architecture with each configuration. These statistics concern (1) the execution, i.e. the *real* time taken by the component to execute one cycle, including preemptions, and (2) the periodicity, i.e. the time between two successive executions.

These statistics show that the periodicity of the tasks are respected in every configuration. Execution times show that:

- ROS tasks are clearly preempted by the disturbing process: their execution time is around 20 and 35ms. each instead of the specified 10 and 20ms;
- Orocos and MAUVE respect the priorities of tasks: real-time tasks are not preempted by the disturbing process, and B is always executed in priority; note that the small variability in execution time comes from the implementation of the benchmark that loops over `getrusage` which is not very accurate.

In this benchmark, the ROS API has been used in its simplest form. The objective is indeed to compare the features provided by the middleware. ROS provides no real-time API. It is nevertheless possible to use native API (e.g., POSIX API) to define real-time threads associated to a node (by setting the scheduler and the priority) but these are standard C++ programming features which suffer a lot of drawbacks regarding modularity/composability of software architectures: limited portability, configuration is hard-coded in the components, no model of the real-time behavior. ROS also provides a *nodelet* API to implement multithreading within nodes. However, this API allows to process data with multiple threads, but does not allow to implement periodic processings. These statistics does not aim to show that ROS is non real-time, but to emphasize what a real-time middleware could bring with respect to real-time guarantees.

4.3 Evaluation of the impact of deployments

Table 3 shows statistics on the first execution of components.

ROS deploys A and B as two separate processes, whose initialization is time consuming (around 50ms for launching node B). MAUVE deployments are consistent with the theoretical deployment presented in Fig. 4: the clock of components are synchronized, and B executes before A. The measure of $t_1 - t_0$ is around 110µs, which is quite efficient. Orocos deployment are unpredictable, whatever the configuration of the deployer thread (more or less priority than components, isolated on a specific core). To explain the high value of the standard deviation of the execution offset of A, we analyzed the execution traces and we have distinguished three kind of behaviours.

In Fig 6, we see that A is started first (first vertical arrow), but it takes some time until A thread is released (at the OS level). Then A starts its execution. When B is released, it preempts A. In this situation, the release dates of A and B follow the starting order (A before B), then B will systematically preempt the execution of A.

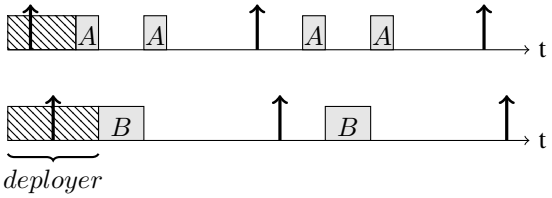

Fig. 6: Orocos deployment, first behaviour: B preempts A

In the situation of Fig. 7, the releases dates of A and B have been switched: B always asks to be executed before A. This situation may happen either when the deployer has a higher

	Execution time				Periodicity			
	A		B		A		B	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev
ROS	35.358	2.561	19.912	1.813	100.002	0.628	100.002	0.658
OROCOS-bg	31.936	1.447	12.028	1.301	99.998	0.831	100.0	0.057
OROCOS-prio	30.097	2.076	12.015	1.276	99.992	2.299	100.0	0.214
OROCOS-isolated	31.573	1.598	12.021	1.194	99.991	1.521	100.0	0.244
MAUVE	20.059	0.725	11.975	1.254	99.997	1.589	100.0	0.216

TABLE 2: Statistics on component executions in milliseconds. The execution time is the time between the start and the end of each execution (including preemptions). The periodicity is the time between two successive executions.

	First execution offset				First response time			
	A		B		A		B	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev
ROS	7.024	1.280	50.272	5.196	37.826	3.040	70.141	6.641
OROCOS-bg	101.332	4.504	100.002	0.013	134.281	0.909	114.324	0.901
OROCOS-prio	104.143	6.603	100.007	0.011	135.018	2.306	114.200	1.120
OROCOS-isolated	105.486	6.920	100.033	0.036	134.154	1.500	113.751	1.278
MAUVE	14.288	1.086	0.110	0.030	36.242	2.373	14.232	1.080

TABLE 3: Statistics on first execution of each component in milliseconds. The execution offset is the time between the start/launch of a component and its first execution. The reponse time is the time between the start and the end of the first execution.

priority (it then prevents A of executing), or if the Orocos internal mechanisms take some time to release A.

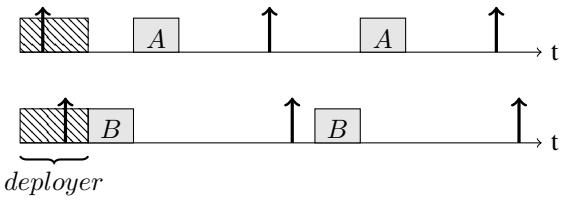

Fig. 7: Orocos deployment, second behaviour: B precedes A

Figure 8 shows a third behaviour where the release dates are such that A is released first, then preempted by B. However, at the first execution, the deployment disturbs this behaviour, making B execute before A. It may happen when the deployer has a higher priority, or if the Orocos deployment takes some time.

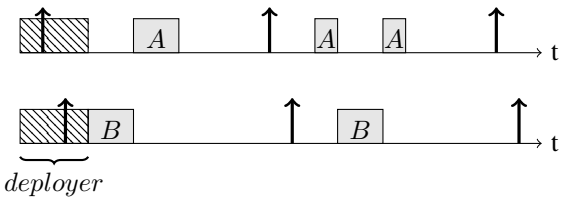

Fig. 8: Orocos deployment, third behaviour: B preemts A except at the first execution

The conclusion of these situations is that the Orocos deployment is not predictable. This non determinism of the deployment, and consequently of the order of releases (and their relative delays), makes the execution model unknown, as stated in the introduction (Sec. 2.1). To manage this uncertainty, we had to make some assumptions on the scheduling of components that lead to overestimated response time computations [8].

4.4 Evaluation of reconfigurations

In order to evaluate the implementation of reconfiguration mechanisms, we modified the MAUVE architecture of the benchmark to make component B reconfigure component A at some time during the execution. The reconfiguration consists in changing A's core by a core that takes a different computation time. The results of this evaluation are shown in Tab. 4.

	Reconfiguration duration		Periodicity of A	
	mean	stdev	mean	stdev
MAUVE	0.196	0.016	99.999	0.057

TABLE 4: Statistics of the reconfiguration benchmark in milliseconds. The reconfiguration duration is the time between the beginning and the end of reconfiguration by component B. The periodicity of A is the time between the beginning of the execution just before the reconfiguration, and the execution directly after the reconfiguration.

These results are consistent with the theoretical reconfiguration model of Fig. 5. The duration of R is about 200µs.

In order to test the robustness of the real-time synchronization when the reconfiguration takes some time, we artificially added the computation time when reconfiguring component A. This situation may happen when the configuration of a component needs to access a sensor driver for instance, or initialize large data structures. Figure 9 shows an actual execution of such a reconfiguration.

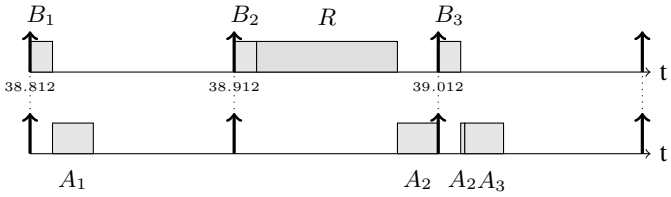

Fig. 9: Actual trace from a long reconfiguration run. Time is given in seconds since the start of the run.

After the reconfiguration is finished, A executes, but is then preempted by B. Execution A_2 then continues in the next period, then A is executed again (A_3). This last execution corresponds to the period of A released at time 39.012. The sequel of the execution then follows the same pattern than for A_1, B_1: B is always executed first, then followed by A.

4.5 Evaluation of the inter-components communication

The MAUVE Runtime is a component-based middleware in which components can exchange data through port by using the different services provided by resources. In this section, we evaluate the time taken to exchange data through the `SharedData` resource[3], that provides `read` and `write` services.

To evaluate the time taken to `read` and `write` a data, we have defined a unique component connected to a `SharedData` resource. The components behavior is to write a data to the resource, wait $10ms$, read the data from the resource, wait $10ms$ and loops on writing again. We have made different runs, in which we changed the size of the data. The results of this evaluation are shown in Tab. 5 for the `write` service and Tab. 6 for the `read` service.

size (octets)	1	10	100	1000	10000	100000
Min (ns)	2088	1991	2078	2670	7365	77571
Max (ns)	7856	8983	10601	11947	35358	151525
Avg. (ns)	4239	4034	4319	4804	12874	120836

TABLE 5: `write` service computation time.

size (octets)	1	10	100	1000	10000	100000
Min (ns)	2042	1955	2167	2182	4705	38280
Max (ns)	11005	11475	10961	10939	18087	105324
Avg. (ns)	4037	3997	4198	4367	9174	64544

TABLE 6: `read` service computation time.

These results show three important characteristics of the MAUVE inter-components communication:

1) the components communication through resources is really efficient (for example reading/writing a data of $1Ko$ takes less than $0.01ms$);
2) for the most common data (size lower than $1ko$) the read and write services take approximatively the same computation time;
3) if the data size is huge the communication can increase[4]. In this case, an efficient solution would be to reimplement a specific *Resource* with services to manipulate the data while preventing from copying large data.

5 Robotic application

To illustrate our proposal, we consider a classical robotic navigation task. The robot (illustrated on Fig. 10) has to move in its environment to reach a specific goal. In this simple example, the environment is represented by a static map previously obtained during an exploration phase.

Fig. 10: Outdoor mobile robot used for the experimentations.

5.1 Implementation of the navigation task

The software architecture is described in Fig. 11. In this figure, components are represented with rounded boxes depicting the internal Finite State Machine and the resources are represented with square boxes. Ports associated with component or resource shells are depicted with dark arrows and resource services with light arrows.

5.1.1 Description of the components

The robot is equipped with an IMU (Inertial Measurement Unit), a GNSS (Global Navigation Satellite System) receiver used for the robot localization and 2D LIDAR (Light Detection

3. `SharedData` is the most commonly used resource in our architectures

4. due to memory access

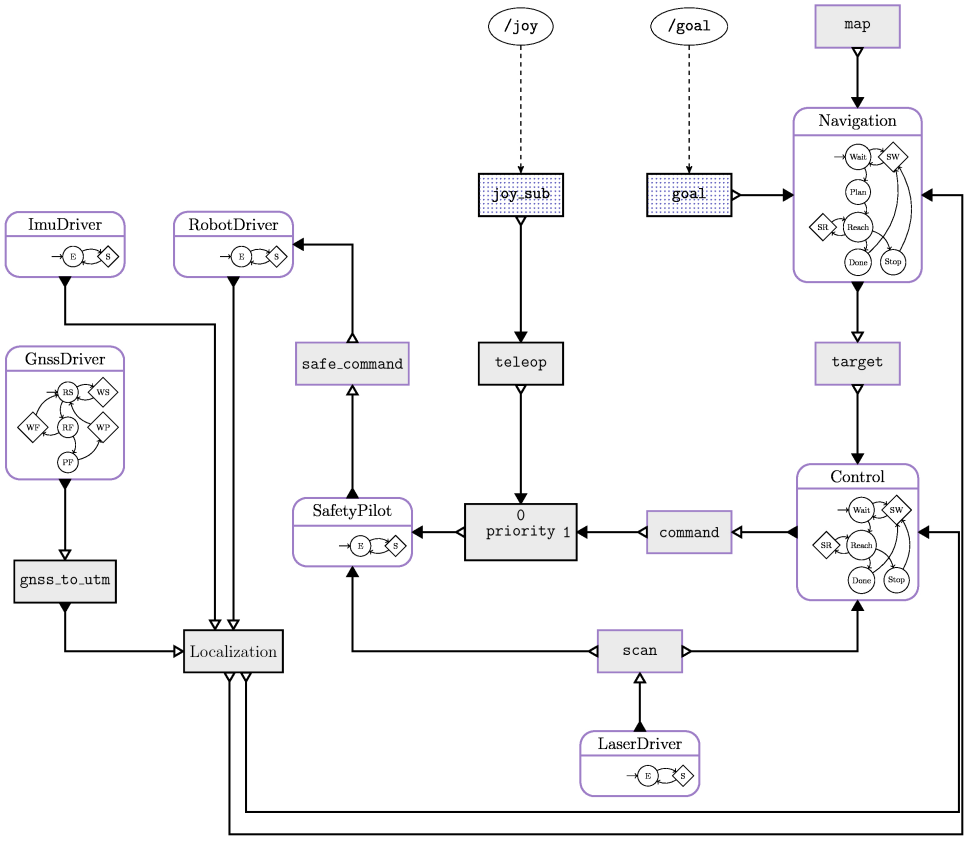

Fig. 11: Architecture of the robotic application

and Ranging) used as a safety sensor to implement obstacle avoidance function. Each sensor is managed by a MAUVE component (`ImuDriver`, `GnssDriver`, `LaserDriver`).

The `RobotDriver` component is used to control the robot motion, it takes velocity commands as input and returns the position of the robot computed from odometers.

The `Control` component implements a guidance law whose objective is to reach a *target* position by avoiding potential obstacles. It takes as inputs the target and current robot positions and uses Laser *scan* from LIDAR to detect obstacles. It computes a velocity *command* used to control the robot motion. In this application, we implemented two different algorithms, through two different cores. The `PotentialFieldCore` (see Listing 2) implements the algorithm of [19]. The `FiniteTimeAvoidanceCore` implements the algorithm of [20]. The former is more suited to cluttered environments with close goals. The latter is more suited to outdoor environments with distant goals.

The velocity command is not directly applied to the `RobotDriver`. In order to insure safe control of the robot, the `SafetyPilot` component takes desired velocity as input and checks the absence of obstacle from laser scan to send a `safe_command` to the robot driver.

The `Navigation` component takes a `goal` position and a `map` as inputs; its role is first to compute a path from the current robot position to the goal and then consecutively send each point of the path as a target to the controller.

The clock-based Finite-State-Machine model proposed by the MAUVE Runtime has been used in several components to implement advanced behaviors. The `Control` component and the `Navigation` component FSM are quite similar, and contain a state dedicated to wait for new goals, and then a state performing the control loop, in which the `Control` component sends velocity commands to the robot, while the `Navigation` component sends waypoints to the `Control` component. The `Control` FSM is shown in Listing 3.

Clock-based FSM also allowed to define behaviors adapted to parsing frames coming from sensor drivers. It is the case of the `GnssDriver`, whose FSM will be further detailed in Sec. 5.2.

5.1.2 Description of the resources

The components exchange data through standard resources of type `SharedData` provided by the MAUVE Runtime: `target`, `command`, `safe_command`, `scan` and `map`.

We have implemented specific resources to define more advanced behaviors. The first type are resources with processing functions that modify input data. In the presented

architecture, such resources are `gnss_to_utm` and `teleop`. The former transforms GNSS position (longitude, latitude) to local coordinates (x, y) using UTM (Universal Transverse Mercator) projection model. The latter transforms joystick state information to a velocity command compatible with the controlled robot. Both manage data input through a `ReadService` and provide processed data as `WritePort` that can trigger another resource.

The second type of specific resource is represented by the `priority` resource whose objective is to select source of data with respect to its priority. In this example, the `teleop` data has a higher priority than the `command` data, enabling the user to take control of the robot at anytime.

The third type of specific resource is `Localization` that takes multiple sensors as input to compute estimated position through a specific filter. This resource will be detailed in Sec. 5.3.

The last one is the *ROS Subscriber* resource that provides an interface with the ROS middleware. In the example, the joystick driver is a ROS node that publishes on topic `/joy` and is used as input for the `teleop` resource. The navigation goal is also specified as a topic on ROS and used as input of the `Navigation` component.

5.2 FSM of `GnssDriver`

Drivers are the software counterpart of hardware that manage physical systems (sensors, actuators, ...). Hardware implementation usually involves hard timing constraints to the software that can be managed thanks to a clock-based FSM. To illustrate this aspect, we detail the FSM of the `GnssDriver` component depicted in Fig. 12.

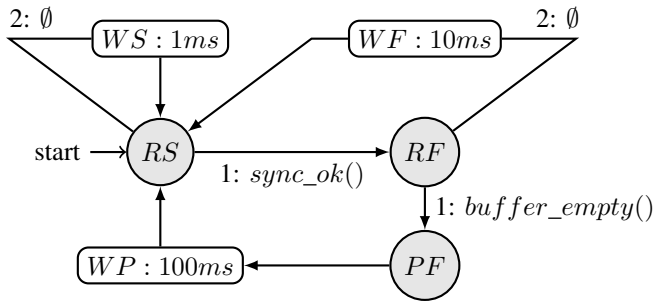

Fig. 12: Clock-based FSM of the *GnssDriver* component: circles represent execution states, W_i are wait states.

The `GnssDriver` provides data periodically through serial port. But the period is not guaranteed when the estimation algorithm embedded in the GNSS receiver takes more time than expected, further the data frame can be corrupted. To deal with this last point, manufacturers provide a standard data frame using header, check-sum and synchronization stamp.

The driver processing is done in three phases: first we look for the synchronization stamp (RS), then we read the frame (RF), and last we check if the data buffer is empty. In this example, the GNSS data is gathered at a frequency of 10 Hz (100 ms period). In case of failure when reading the synchronization stamp, the FSM goes to state WS and waits for 1 ms. If an error occurs when reading the frame (state RF), the FSM goes to state WF and wait for 10 ms before trying to read a new frame, otherwise it goes to state PF to parse the frame. Finally, if the reading has succeeded (going successively through states RS, RF, PF), a wait is done according to the period of the sensor (here 100 ms).

The benefits of the approach are double: first it provides a cleaner implementation than a monolithic function, and second it allows to deal with data polling in a more efficient way from the timing point of view.

5.3 Resource `Localization`

The `Localization` resource deals with fusion of sensors to provide estimate of the robot position relative to a reference frame. It is based on a common Extended Kalman Filter which roughly processes data in two phases: a prediction phase that uses internal model to estimate the position in the future at a given instant (t), and a correction phase (update) that uses measured data to update the current estimate of the position. These two phases take into account the co-variance associated both to the prediction (model co-variance) and to the measures (sensors co-variance). The main advantage of this approach is to integrate different types of sensor thanks to measure function.

The Kalman filter does not need a periodic activity because the prediction can be done for different components running at different periods and the sensors provide data in an asynchronous way. Thus, it is more convenient to implement the Kalman filter as a *resource* such that predict and update functions are respectively `ReadService` and `WriteService` associated to `ReadPort` and `WritePort` of components. Listing 7 details the interface of this resource. `Update` services are specific to a type of data produced by a sensor, e.g., IMU, GNSS (pose), Odometer (twist). `Predict` services can be called to provide estimate at a given time or implicitly at the current time.

This implementation provides a generic Kalman filter that can be adapted to different configuration of sensors when instantiating the *architecture*.

5.4 Reconfiguration capabilities

This architecture provides some possibilities of reconfiguration. The first available reconfiguration consists in switching the core of the `Control` component. This component has two Core's implementations, as mentioned in Sec. 5.1.1: an implementation more suited to cluttered environments, and an implementation more suited to open environments with more distant goals, typically in outdoor settings. The reconfiguration could be triggered by a supervision component that would

Listing 7: EKF Resource

```
1
2  struct EkfCore: public Core<EkfShell>
3  {
4    bool configure_hook() override;
5    void cleanup_hook() override;
6
7    void update_imu(Imu m);
8    void update_pose(Pose m);
9    void update_twist(Twist m);
10   MotionState predict();
11   MotionState predict_at(double t);
12 };
13
14 struct EkfInterface: public Interface<EkfShell,EkfCore>
15 {
16   WriteService<Imu> & update_imu =
17     mk_write_service<Imu>("update_imu",
18                           &EkfCore::update_imu);
19
20   WriteService<Pose> & update_pose =
21     mk_write_service<Pose>("update_pose",
22                            &EkfCore::update_pose);
23
24   WriteService<Twist> & update_twist =
25     mk_write_service<Twist>("update_twist",
26                            &EkfCore::update_twist);
27
28   ReadService<MotionState> & predict =
29     mk_read_service<MotionState>("predict",
30                                  &EkfCore::predict);
31
32   CallService<double,MotionState> &predict_at =
33     mk_call_service<MotionState,double>("predict_at",
34                                  &EkfCore::predict_at);
35 };
```

switch the `Control` core based on the area where the robot is currently evolving.

The second available reconfiguration consists in disconnecting the `GnssDriver` component and the `Localization` resource. This may be useful for instance when the GNSS data becomes unreliable, which could be monitored by using the number of available GNSS satellites, or the dispersion of the GNSS signal. Disconnecting the GNSS data may avoid the localization filter to integrate unreliable data, which may disturb the estimation process. Once more, this reconfiguration will be activated by a supervision component that monitor the status of the GNSS data, and disconnect the component and the resource, or unconfigure the `gnss_to_utm` resource to cut the data processing.

The management of the different reconfiguration mechanisms will be implemented in a generic supervision module that will be designed in our future works. For these experimentations, the reconfiguration is triggered manually by the operator.

6 Conclusion

In this paper, we have presented the MAUVE Runtime, a middleware for designing robotic software architectures. MAUVE Runtime main focus is to provide real-time guarantees on the execution. The MAUVE Runtime moreover provides reconfiguration mechanisms, such that the real-time execution model is perfectly specified. We have presented the different elements of the MAUVE Runtime, that defines software architectures by connecting components and resources. Components have a real-time activity defined by a clock-based finite state-machine, that allows to model complex timed behaviors. Resources are used to provide data sharing, possibly including some processing functions. We have then evaluated the implementation of the MAUVE Runtime with a benchmark, compared to ROS and Orocos architectures. This benchmark has first shown the usefulness of a real-time middleware to guarantee a real-time behavior, and second that MAUVE deployment behavior is compliant with its specification. As far as we know, none of the existing middlewares used in robotics provides *real-time safe* reconfiguration mechanisms as defined and implemented in the MAUVE Runtime.

We finally presented the architecture of a ground robot performing a navigation task, and we have illustrated how the several features of the MAUVE Runtime allows to implement advanced features and behaviors within the architecture. In this application, the activation of the reconfiguration (and the monitoring that will trigger this reconfiguration) are not managed by the runtime itself. Current work consists in adding in the MAUVE Runtime the concept of *architecture monitors*, in order to identify which components will be able to reconfigure the architecture.

The MAUVE Runtime and associated documentation and libraries are available at

https://mauve.gitlab.io

References

[1] D. Brugali and P. Scandurra, "Component-Based Robotic Engineering (Part I): Reusable building blocks," *IEEE Robotics and Automation Magazine*, vol. 16, no. 4, 2009. [Online]. Available: https://doi.org/10.1109/MRA.2009.934837 2

[2] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II): Systems and models," *IEEE Robotics and Automation Magazine*, vol. 17, no. 1, 2010. [Online]. Available: https://doi.org/10.1109/MRA.2010.935798 2

[3] A. Basu, M. Gallien, C. Lesire, T.-h. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis, "Incremental Component-based Construction and Verification of a Robotic System," in *Eureopean Conference on Artificial Intelligence (ECAI)*, Patras, Greece, 2008. [Online]. Available: https://doi.org/10.3233/978-1-58603-891-5-631 2.1

[4] A. Mallet, C. Pasteur, and M. Herrb, "GenoM3: Building middleware-independent robotic components," in *International Conference on Robotics and Automation (ICRA)*, Anchorage, AK, USA, 2010. [Online]. Available: https://doi.org/10.1109/ROBOT.2010.5509539 2.1, 3.4

[5] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Darmstadt, Germany, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-17319-6_31 2.1

[6] A. Steck and C. Schlegel, "Towards quality of service and resource aware robotic systems through model-driven software development," in *International Workshop on Domain-Specific Languages and models for Robotic systems (DSLRob)*, Taipei, Taiwan, 2010. [Online]. Available: https://arxiv.org/abs/1009.4877 2.1

[7] N. Gobillot, C. Lesire, and D. Doose, "A Modeling Framework for Software Architecture Specification and Validation," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Bergamo, Italy, 2014. [Online]. Available: https://doi.org/10.1007/978-3-319-11900-7_26 2.1

[8] N. Gobillot, D. Doose, C. Lesire, and L. Santinelli, "Periodic state-machine aware real-time analysis," in *IEEE Conference on Emerging Technologies & Factory Automoation (ETFA)*, Luxembourg, Luxembourg, 2015. [Online]. Available: https://doi.org/10.1109/ETFA.2015.7301403 2.1, 4.3

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, Kobe, Japan, 2009. [Online]. Available: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf 2.1

[10] 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. [Online]. Available: https://doi.org/10.1109/ROBOT.2005.1570129 2.1, 3.4

[11] "ROCK, the Robot Construction Kit." [Online]. Available: http://www.rock-robotics.org 2.1

[12] D. Brugali and L. Gherardi, "HyperFlex: A model driven toolchain for designing and configuring software control systems for autonomous robots," *Studies in Computational Intelligence*, vol. 625, pp. 509–534, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-26054-9_20 2.1

[13] N. Gobillot, F. Guet, D. Doose, C. Grand, C. Lesire, and L. Santinelli, "Measurement-based real-time analysis of robotic software architectures," in *International Conference on Intelligent Robots and Systems (IROS)*, Daejeon, South Korea, 2016. [Online]. Available: https://doi.org/10.1109/IROS.2016.7759509 2.1

[14] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Transactions on Software Engineering*, vol. 23, no. 12, 1997. [Online]. Available: https://doi.org/10.1109/32.637390 2.2

[15] Z. Yu, I. Warren, and B. Macdonald, "Dynamic reconfiguration for robot software," in *International Conference on Automation Science and Engineering (CASE)*, Shanghai, China, 2006. [Online]. Available: https://doi.org/10.1109/COASE.2006.326896 2.2

[16] X. Yu, K. Niyogi, S. Mehrotra, and N. Venkatasubramanian, "Adaptive middleware for distributed sensor environments," *IEEE Distributed Systems Online*, vol. 4, no. 5, 2003. [Online]. Available: https://doi.org/10.1109/MDSO.2003.1 2.2

[17] J. M. Cobleigh, L. J. Osterweil, A. Wise, and B. S. Lerner, "Containment units: A hierarchically composable architecture for adaptive systems," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002. [Online]. Available: https://doi.org/10.1145/605466.605491 2.2

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computing*, vol. 39, no. 9, 1990. [Online]. Available: https://doi.org/10.1109/12.57058 3.5, 3.8

[19] M. Guerra, D. Efimov, G. Zheng, and W. Perruquetti, "Avoiding local minima in the potential field method using input-to- state stability," *Control Engineering Practice*, vol. 55, 2016. [Online]. Available: https://doi.org/10.1016/j.conengprac.2016.07.008 5.1.1

[20] ——, "Finite-time obstacle avoidance for unicycle-like robot subject to additive input disturbances," *Autonomous Robots*, vol. 41, no. 19, 2017. [Online]. Available: https://doi.org/10.1007/s10514-015-9526-0 5.1.1

David Doose received his M. Sc. degree in Computer Sciences from University Paul Sabatier, Toulouse, France in 2002, and the Ph. D. degree in Computer Sciences from University of Toulouse, France, in 2006. In 2007, he was a post-doctoral fellow at University of Toulouse in collaboration with Airbus group. Since 2007, he has been a research fellow at ONERA, in the Computer Science department. His main topics of interest include embedded systems, real-time analysis, formal methods and recently software engineering for robotics.

Christophe Grand received his M. Sc. degrees in mechanical engineering from ENSAM, Paris, France in 2000 and the Ph. D. degrees in Robotics from the University Pierre et Marie Curie (UPMC), Paris, France in 2004. From 2005 to 2013, he was associate professor at UPMC, in the Robotics Department. Since 2013, he is a research follow at ONERA, in the System Control Department. His research interest covers robotics, embedded systems and multi-robot cooperation.

Charles Lesire received his M. Sc. degree in Computer Sciences from ENAC, Toulouse, France in 2003, and the Ph. D. degree in Computer Sciences from University of Toulouse, France, in 2006. In 2007, he was a post-doctoral fellow at LAAS-CNRS, in the robotics team. From 2008 to 2016, he was a research fellow at ONERA in the System Control department (DCSD). Since 2017, he has been a research fellow at ONERA in the Information Processing department (DTIS). His main topics of interest include software engineering for robotics, embedded decision-making, and cooperation of heterogeneous robots. He is a member of the IEEE RAS Technical Committee on Software Engineering for Robotics and Automation (TC-SOFT).