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

David Doose and Christophe Grand and Charles Lesire
ONERA – The French Aerospace Lab, Toulouse, France
Email: david.doose@onera.fr, christophe.grand@onera.fr, charles.lesire@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 preliminary design of a new component-based middleware that allows to perform software architecture reconfigurations with a focus on real-time constraints.**

## I. 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. To guaranty software reliability, we have proposed an approach sit on three fundamental pillars: (1) the modeling of software architecture, (2) its implementation on a deterministic execution middleware, conforming to the model, (3) the analysis of real-time constraints [1].

Many middlewares have been proposed in the robotics community like ROS, Orocos or Yarp (see [2] for a survey). Some of them focus on real-time execution, others on modularity and scalability or on inter-process communications. The originality of our approach is the design of a toolchain that allows a schedulability analysis of the software based on models, from which executable code is generated. In our previous works, Orocos, a real-time component-based middleware [3], has been used as the generation target to evince the benefit of the toolchain [4]. However, previous experiments have exhibited some lacks of Orocos with respect to the control of task synchronisation, and on reconfiguration features. The ability to reconfigure the software architecture inline is an essential feature in order to cope with failures or disturbances. Indeed, some faults can be addressed with a redundancy in sensing or acting processing chains, requiring sound and efficient reconfiguration capabilities to adapt the software architecture. Although it may be possible to reconfigure some flows in existing middlewares [3], [5], [6] (i.e. to rewire the ports connections), none of these middlewares also supports reconfiguration of the computational part of components, nor provides real-time mechanisms for these reconfigurations.

In this paper, we present the preliminary design of a new middleware that addresses the lacks identified so far, while keeping the best practice concepts from our previous works.

## II. Background and Motivations

### A. Illustrative example

To illustrate our proposal, we consider a multi-robot exploration mission, in which each robot has an architecture as described in Fig. 1. The *Robot* component takes velocity

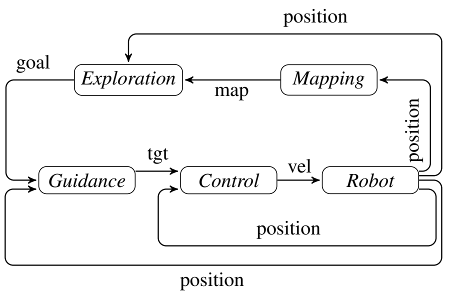

Fig. 1: Architecture of the exploration mission

commands as input and returns the 2D position of the robot. The *Control* component implements a PID controller that computes a velocity command based on the current robot position and a target position. The *Guidance* component takes goal positions as input; 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 *Mapping* component takes the position of the robot to update the status of a map, tagging each cell as explored or unknown. The *Exploration* component computes the next point to explore; it exchanges data with other robots in order to allocate exploration areas to each robot. In case of a communication failure, we want to be able to continue the mission. We then need to reconfigure the architecture, by replacing the *Exploration* algorithm by a *Local Exploration*, where the robot takes decision on the points to explore on its own, without communicating with the other robots.

### B. Background

The previously proposed MAUVE toolchain [1] was based on a three-step development process: (1) modeling the architecture, (2) generating the corresponding code, instrumented with tracing tools, (3) analyze real-time behavior based on the model and the runtime execution traces.

The modeling step was based on component-based architecture models using the MAUVE Domain Specific Language (DSL). In this language, the software architecture is described as components connected to each others through directed data ports and configured through properties. The behavior of each component is defined by a finite state machine whose activity is managed by a dedicated real-time task. Components are instantiated and their data ports connected together to form the system architecture which is initialized and launched by a specific tool: the deployer.

The software safety insurance is based first on the automatic generation of laborious and systematic sections of code, and secondly on the generation of runtime execution traces that are combined with the model to verify the schedulability of the whole architecture for a specific target [7], [4].

To summarize, our approach is based on three foundations: (1) the MAUVE DSL, (2) an efficient and reliable runtime middleware (that conforms to the model), (3) real-time analysis. For the second, we chose the well-know real-time middleware Orocos [3] which offers interesting features such as the finite state machine description of components behavior, the thread-safe (lock-free) communication paradigm and the interoperability with ROS. As explained below, this choice of Orocos imposes a specific runtime model which proved not to be always adapted to our requirements. Other middlewares such as ROS [5] do not provide any runtime model, meaning that we must define a model anyway.

## C. Periodic State Machines

In the previous execution model [7], each component behavior is defined by a periodic finite-state machine (PSM), where each state executes a specific action. Each component is then mapped to a real-time periodic task, and executes a single action during each period. Let's consider the *Guidance* component of the architecture of Fig. 1. The PSM of the *Guidance* component is shown in Fig. 2. The component is

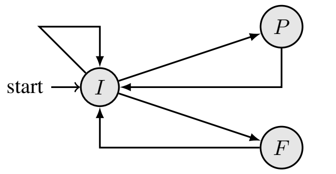

Fig. 2: PSM of the *Guidance* component

initially in the *Idle (I)* state, where it reads data in its input ports. If a new goal is received, the component goes to the *Planning (P)* state, where it computes a path to reach the goal. If no new goal is received but a path is being executed, the component goes to the *Following (F)* state, in which it checks if the current point has been reached, and then send the next one. This example highlights two limitations of PSMs:

- we cannot execute two actions in a single period, and we have to wait one period between executing the *I* and *F* states, whereas it would be more efficient to execute both actions in the same period;
- we can allocate only one period to each component, meaning that the component will have the same reactivity in the *F* and *P* states; as path planning takes more computation time, it would be more interesting to have a larger period when in the *P* state, and a smaller in the *F* state to be more reactive.

## D. Resources

In components-based architectures, data need to be exchanged between components. Real-time processed information like sensor measures or actuator control inputs are naturally transported through directed data ports (as done in Orocos). As this approach needs local copy of the data to protect their access, it is not adapted for more complex data structures like environment maps or system databases which rather need a get/update access paradigm.

All these data exchange mechanisms can be seen as a resource sharing problem, and then need to be generalized. Further, this generalization can constitute a convenient way to implement interoperability with other middlewares by including their specific data transport inside specific resources. This avoids to implement particular functions within components that do not have a specific real-time activity.

## E. Tasks synchronization

One of the drawbacks raised is the lack of determinism in the deployment process. Let's consider a simple example made of two tasks ($\tau_1$, $\tau_2$) executed on a single processor; the deployer objective is to "start $\tau_1$ and start $\tau_2$". Timeline of Fig. 3 illustrates the actual execution. The deployer is launched at $t_0$, and at $t_1$ task $\tau_1$ is released. Contrary to the deployer, $\tau_1$ is a real-time thread; consequently the deployer is preempted by $\tau_1$ and has to wait the end of $\tau_1$ execution before starting task $\tau_2$.

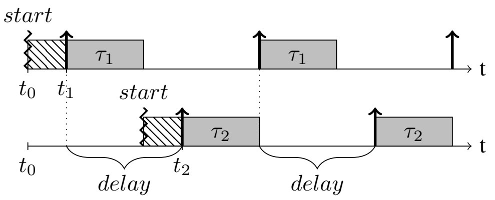

Fig. 3: Synchronization lost

This timeline highlights three major problems:

- even if the deployer starts at $t_0$ the release date of task $\tau_1$ is $t_1$ and the delay between $t_0$ and $t_1$ is unknown;
- it is impossible to synchronize tasks $\tau_1$ and $\tau_2$;
- the delay between $t_1$ and $t_2$ is unknown.

These problems lead to limitations in the deployment capabilities and to pessimism in the schedulability analysis [7].

### F. Inline reconfiguration

Reconfiguring the architecture inline is a key feature to improve the capability to adapt to unforeseen events. Basic reconfigurations, like changing the computational part of a component, are necessary to switch between algorithms (e.g., to manage several control modes according to some actuator faults) efficiently. Complex reconfigurations, where some components are added or removed, and connections are re-wired, are necessary when new processing chains have to be set up, e.g. when a sensor is broken (and a new one has to be used instead), or becomes useless (e.g., GPS when indoors).

These reconfigurations must be performed rigorously, especially regarding real-time synchronisation in order to master the impact of reconfigurations on the reactivity of the whole application. We then need to have a deterministic synchronisation scheme between tasks, and to be able to reset this synchronisation after a reconfiguration.

## III. MAUVE Runtime

In this section, we describe the concepts behind the design of the MAUVE runtime, and we present some details regarding its implementation as a C++ library.

### A. Component

A component is defined by three entities: a *shell*, a *core* and a *finite-state machine* ($FSM$). The *shell* describes the interface of the component. It contains *ports* and *properties*. The *shell* is the only visible part outside the component. The *core* describes both the algorithmic part and the inner data of the component. Thus, the *core* defines attributes and methods. The *core* depends on the *shell* which means its methods can interact with the *shell*. The *FSM* describes the behavior of the component. The different actions in the *FSM* rely on the core methods. Listing 1 shows the definition of the shell of the *Guidance* component. It contains two properties, two input ports and one output port.

Listing 1: Shell of the *Guidance* component

```
struct GuidanceShell : public Shell {
  Property<double>   & target_threshold;
  Property<double>   & path_step;
  ReadPort<Point2D>  & position;
  ReadPort<Point2D>  & goal;
  WritePort<Point2D> & target;
};
```

Listing 2 defines the core of the *Guidance* component. 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 FSM actions.

### B. Enhanced Finite State Machine

Lessons learnt from PSM modeling using the MAUVE DSL have brought us to enhance the specification of FSMs. New clock-based FSMs are composed of two different type of states:

Listing 2: Core of the *Guidance* component

```
struct GuidanceCore : public Core<GuidanceShell> {
  Point2D current_goal , current_position;
  std::queue<Point2D> path;
  bool has_new_goal();
  void compute_path();
  void follow_path();
};
```

- *execution* states are aimed to execute code (i.e. methods defined is the component's core).
- *wait* states *pause* the component for a specific duration.

There is no restriction in the clock-based FSM structure, thus an execution state can be followed by another execution state.

Figure 4 represents the clock-based FSM of the *Guidance* component. The limitations of the original PSM (Fig. 2) are

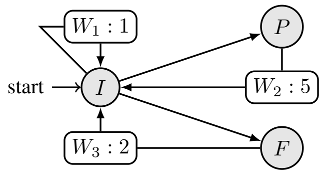

Fig. 4: Clock-based FSM of the *Guidance* component: circles represent execution states, $W_i$ are wait states.

addressed in the following way: (1) the execution state $I$ is directly connected to $P$, then they can be executed without any delay; (2) wait states $W_i$ have different clock values, making the cycle $I \to P \to I$ being executed with a period of 5 time units, $I \to F \to I$ with a period of 2, and $I \to I$ with a period of 1.

### C. Resources

In MAUVE runtime, we introduce a generalized inter-components communication scheme with the notion of *Resource*. A *Resource* has two main objectives: first it contains a data, second it provides different services to interact with its inner data. Component ports are connected to a resource using its corresponding services. The basic resources `SharedData` and `RingBuffer` are provided by the runtime to mimic classical dataflow communication, and are accessible through read and write service.

Moreover, this approach allows developers to define their own resources by inheriting from the **Resource** class and defining its services. Thus, Resource can be implemented to offer access to complex or large data structures through a get/update access paradigm. Also, by providing in the interface, functions dedicated to data streaming, the interaction with other middleware like ROS can be integrated.

### D. Architecture and Deployment

Components and resources are instantiated and connected within *architectures*. 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. 5 illustrates the deployment of two tasks executed on the same processor by the MAUVE runtime. Tasks $\tau_1$ and $\tau_2$ are synchronized with the deployer, and their synchronization is maintained during all the execution, contrary to the behavior resulting from the Orocos deployer (Fig. 3).

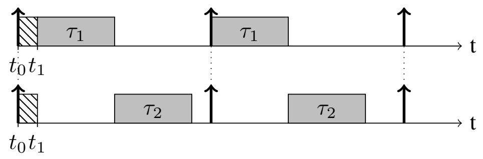

Fig. 5: Tasks synchronization

## E. Reconfiguration

The MAUVE runtime allows to perform reconfigurations by relying both on the component model and on the real-time deployment. Indeed, it is 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 3 shows a code snapshot performing a reconfiguration during the exploration mission (Fig 1, Sec. I). In order to make a sound

Listing 3: Reconfiguration code snapshot: the core of component `exploration` is replaced.

```
time_ns_t clock = exploration_task->get_time();
exploration_task->stop();
exploration->cleanup_core();
exploration->replace_core<LocalExplorationCore>();
exploration->configure_core();
exploration_task->start(clock, nullptr);
```

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 *Exploration* component. We then configure this core and restart the task according to the same clock reference. Figure 6 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 the $R$ activity). The task is restarted, the synchronization being maintained.

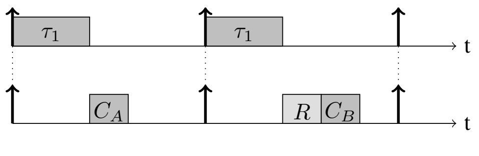

Fig. 6: Task synchronisation during reconfiguration of a core

## F. Implementation

The implementation of the runtime concepts must guarantee a real-time and deterministic execution of the components. In that purpose, we made the following implementation choices:

- the system is executed on a real-time Linux OS;
- each component is executed in a real-time posix thread;
- the priority inheritance protocol [8] is used to access common data;
- the monotonic clock is used as the reference for tasks;
- posix barrier are used to synchronize tasks at deployment.

## IV. Roadmap

The MAUVE runtime development is being finalized and evaluated through the illustrative example. The ongoing work mainly concerns the integration on a complete modeling workflow, in which the planned developments are:

- to update the DSL and code generation using the MAUVE runtime;
- to integrate generation of tracing in order to plug with the real-time analysis process;
- to define a workflow to have data type models allowing to generate resource-related code, both to ease development and to automate interoperability with other middlewares;
- to model coordinate frame transformations to enhance semantic associated with physical data.

MAUVE runtime is released as an open-source project, under LGPL licence, as part of the MAUVE toolchain. Regarding the MAUVE runtime, the MAUVE toolchain also provides tutorials, common types for robotic applications, and some basic elements like common shells, state machines of specific task models (e.g., periodic state machines), . . . The MAUVE toolchain is available on gitlab at the address

https://gitlab.com/MAUVE/mauve_toolchain.

## V. ACKNOWLEDGEMETS

This work is partially supported by the CPSELabs project funded by European Community's Horizon 2020 Programme under grant agreement no 644400.

## References

[1] N. Gobillot, C. Lesire, and D. Doose, "A Modeling Framework for Software Architecture Specification and Validation," in *SIMPAR*, Bergamo, Italy, 2014.

[2] P. Iigo-Blasco, F. D. del Rio, M. C. Romero-Ternero, D. Cagigas-Muiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, 2012.

[3] P. Soetens and H. Bruyninckx, "Realtime Hybrid Task-Based Control for Robots and Machine Tools," in *ICRA*, Barcelona, Spain, 2005.

[4] N. Gobillot, F. Guet, D. Doose, C. Grand, C. Lesire, and L. Santinelli, "Measurement-based real-time analysis of robotic software architectures," in *IROS*, Daejeon, South Korea, 2016.

[5] 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.

[6] 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.

[7] N. Gobillot, D. Doose, C. Lesire, and L. Santinelli, "Periodic state-machine aware real-time analysis," in *ETFA*, Luxembourg, 2015.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, 1990.