Robotics and Autonomous Systems 160 (2023) 104322

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

A BPMN-driven framework for Multi-Robot System development

Flavio Corradini [a], Sara Pettinari [a,*], Barbara Re [a], Lorenzo Rossi [a], Francesco Tiezzi [b]

[a] School of Science and Technology, University of Camerino, Italy
[b] Dipartimento di Statistica, Informatica, Applicazioni, University of Florence, Italy

ARTICLE INFO

Article history:
Available online 2 December 2022

Keywords:
Multi-robot systems
BPMN
Model-driven development
Enactment framework

ABSTRACT

Programming robotic systems is often a challenging task requiring advanced skills, especially when the goal is to ensure loosely-coupled coordination in heterogeneous Multi-Robot Systems (MRSs). Model-driven approaches for robotic system engineering have shown their benefits in facilitating the development of robots' behavior, controllers, and system components. However, the state of the art still lacks contributions addressing crucial aspects of the model-driven approach applied to MRSs, such as developing robots' distributed cooperation through models supporting the communication among robots.

In this paper, we present a novel framework for modeling, configuring and enacting the cooperative behaviors of MRSs through collaboration diagrams as provided by the BPMN 2.0 standard. The advantages of our solution lie, indeed, in the use of BPMN, which provides easily understandable and highly expressive diagrams for representing the cooperation among distributed robots, and benefits from a wide list of supporting tools. Starting from the selection of BPMNelements, we define a set of guidelines for driving the developer in modeling an MRS mission using BPMN. The developer configures the resulting collaboration diagram to link elements in the model to the robotic middleware, ROS2 in the toolchain we implemented. Finally, the configured model is enacted by BPMN engines integrated into the ROS2 middleware run by each robot involved in the MRS, thus obtaining a fully distributed cooperation. We assess our framework's effectiveness through experiments in simulated and real environments.

1. Introduction

Nowadays, thanks to the advantages brought into everyday human life, Multi-Robot Systems (MRSs) are widely adopted in many application domains, e.g., agriculture, manufacturing, industry, and health. Indeed, MRSs can reduce the need for (possibly dangerous) human operations (e.g., the cleanup of toxic waste and rescue missions in disaster scenarios) and human errors in repetitive tasks [1]. At the same time, the adoption of MRSs can improve productivity and service quality in human-centered applications [1].

MRSs consist of heterogeneous robots acting on the environment and interacting with each other to accomplish a common goal. A single entity has few functionalities in these systems since the real power lies in the collaboration among robots. Thus, concerning single-robot systems, MRSs can fulfill a task faster and cheaper, guaranteeing scalability, reliability, flexibility, and versatility [2]. Overall, the behavior of single robots and their interactions should produce the cooperative behavior of the whole system without the need for centralized control. Indeed, depending on the behavior of every single robot and on the way they coordinate each other, the robots' mission may ultimately fail, or it may succeed in its accomplishment with different levels of quality, usage of robots, and resources. As an example, consider a scenario where drones have to patrol a warehouse; according to the battery recharge schedule adopted by the drones, different quality levels can be achieved, e.g., it can be ensured the presence of a minimum number of drones in the area to monitor or minimized the number of recharges to reduce the energy consumption.

To foster and make more accessible the development of MRSs, many robotic-centered middlewares provide facilities through which developers can build robotics software abstracting from the underlying hardware. In this paper, we take as reference Robot Operating System (ROS), especially ROS2. It is the most prominent middleware [3] which comes with ready-to-use libraries handling complex tasks, and embeds the Data Distribution Service (DDS) [4] standard protocol both for intra- and inter-robot broadcast communications. Despite the facilities provided by ROS, developing reliable MRSs is still a challenging task [5], which requires advanced programming skills to consider all at once the

* Corresponding author.
E-mail addresses: flavio.corradini@unicam.it (F. Corradini), sara.pettinari@unicam.it (S. Pettinari), barbara.re@unicam.it (B. Re), lorenzo.rossi@unicam.it (L. Rossi), francesco.tiezzi@unifi.it (F. Tiezzi).

https://doi.org/10.1016/j.robot.2022.104322
0921-8890/

distributed computations made by each robot and the message exchanges needed to coordinate them. To cope with this challenge, the literature presents several model-driven approaches for robotic systems [6,7]. Nevertheless, these approaches either lack in providing a high-level abstraction of the whole MRS behavior, as they focus on a single robot mission, or even discard the scenarios involving multiple robots. At the same time, many approaches do not exploit a direct model interpretation, relying instead on a model-to-code translation that prevents a developer from modifying the system's behavior at run-time.

In [8], we evaluated the use of the BPMN standard notation [9], and in particular of the BPMN collaboration diagrams, for modeling effectively MRSs. The result of that study is that collaboration diagrams fit well in representing distributed and cooperative scenarios like MRSs, since they express at a high level of abstraction the control flow and the interactions of different entities.

In this work, we take a step forward in this direction by extending the achievements of [8] with FaMe (BPMN-driven FrAmework for Multi-robot systEms development). FaMe is a framework for **model-driven** development of robots behavior based on **modeling guidelines** and an **enactment methodology** that guarantees system distribution. In a nutshell, the FaMe framework proposes a selection of BPMN elements and a set of modeling guidelines driving the user during the specification of an MRS. Guidelines provide a link between BPMN elements and robotic concepts and foster the modularization of the diagram employing *callable* behaviors to increase the level of abstraction. The resulting collaboration is then configured by the user to be compliant with the robotic middleware (ROS2 in our case), and is automatically split into single executable processes, one for each robot in the MRS. The execution of the processes is fully distributed since every robot embeds a BPMN engine, integrated with ROS, that executes its own process.

The novelty of FaMe lies in a disciplined use of the BPMN standard notation permitting: (i) to represent a scenario with multiple robots using a single collaborative model; (ii) to model distinctive aspects of an MRS without resorting to domain-specific extensions; (iii) to reduce the time-to-code through the re-use of already modeled behaviors; (iv) to control robots behavior through the execution of the modeled collaboration via BPMN semantic engines; and (v) to ensure MRS's resilience thanks to a distributed execution of the mission. To the best of our knowledge, these distinctive features are not considered altogether by any other model-driven approach for MRS programming. To assess the feasibility and the effectiveness of FaMe, we show the framework at work in two different robotics scenarios, both in simulated and real environments, and we measure its performances.

The rest of the paper is organized as follows. Section 2 provides notions concerning the technologies at the basis of our approach and presents a running example depicting a smart agriculture scenario. Section 3 presents the FaMe framework and how it supports the phases of the MRS development. Section 4 describes the FaMe toolchain. Section 5 illustrates the framework at work on the running example and on another robotics scenario, and provides an overview of the framework performances. Section 6 compares our solution with respect to the related works. Finally, Section 7 concludes the paper and touches upon directions for future work.

2. Background notions

This section introduces the background concepts used in the paper to describe the proposed framework. We firstly present the ROS middleware. Then, we give an overview of the BPMN standard and the collaboration diagrams, bounding the discussion to the elements used in FaMe. Finally, we present the smart agriculture scenario we use as a running example throughout the paper.

2.1. Robot Operating System

ROS (www.ros.org) is one of the most famous and used open-source frameworks for programming robots. It provides an abstraction layer on top of which developers can build robotics applications. The first version of ROS, i.e., ROS1, is suitable for developing single robot systems, while in recent years, a second release, i.e., ROS2, has been proposed to overcome ROS1 performance and scalability problems, thus achieving full support for MRSs. ROS2 provides real-time control and exploits a communication mechanism more suitable for inter-robot interactions. This motivates why we take ROS2 as a reference release in this paper. From now on, for the sake of presentation, we will refer to ROS2 simply as ROS.

ROS is designed as a framework of distributed nodes, which are processes able to perform computations and designed to achieve single purposes, e.g., controlling motors or the ultrasonic sensor. Each robot can be seen as a collection of nodes capable of sensing the environment, acting on it, and making decisions. The ROS architecture is illustrated in Fig. 1. Nodes are in a network and can communicate with each other by sending and receiving data via topics. *Topics* exploit a publish–subscribe pattern allowing to perform topic-based communication, where a message published over a topic can be read by any number of other nodes subscribed to that topic. The keystone enabling the development of MRSs in ROS is the OMG standard DDS (www.omg.org/spec/DDS). It enables real-time distributed systems to operate securely as an integrated whole. It introduces a global data space where applications can share information by simply reading and writing data objects. ROS takes advantage of DDS to enable several features. Among them, the one that mostly facilitates the development of MRSs is the distributed discovery system that allows ROS nodes to communicate with each other without the use of a master node. Notably, before deploying the ROS code into real robots, a good practice is to exploit a simulation environment. The reference simulator for ROS is Gazebo (www.gazebosim.org). It permits designing and testing robotic applications rapidly and in a safe, yet realistic, environment. As depicted in Fig. 1, Gazebo is embedded in the ROS architecture as a node, therefore, developers can use the same code for controlling the robots in both real and simulated environments.

2.2. BPMN notation

BPMN is an OMG standard notation that provides different diagrams for modeling business activities. Among others, collaboration diagrams are the most suited for depicting both the behaviors and the topic-based interactions among different parties that have to reach a shared goal [10].

Fig. 2 shows a subset of BPMN elements we selected specifically for the proposed modeling approach from a total of 85 BPMN elements. The considered elements are the result of the investigation conducted in [8] in which we addressed the problem in a top-down fashion via the modeling activity we performed on different application scenarios and, in a bottom-up fashion, via the experiments carried out with ROS implementations via the Gazebo simulator.

We briefly introduce the BPMN elements resulting from this selection and their execution semantics. A collaboration diagram consists of a collection of (possibly multi-instance) *pools*, each of which contains a process, see Fig. 3 for an example. A *pool* is a logical representation of an entity that takes part in the collaboration. Graphically, *pools* are rectangles containing other BPMN elements that compose process diagrams. They are labeled with the name of the entities they represent. A pool with a multi-instance marker (three vertical lines) depicted near the

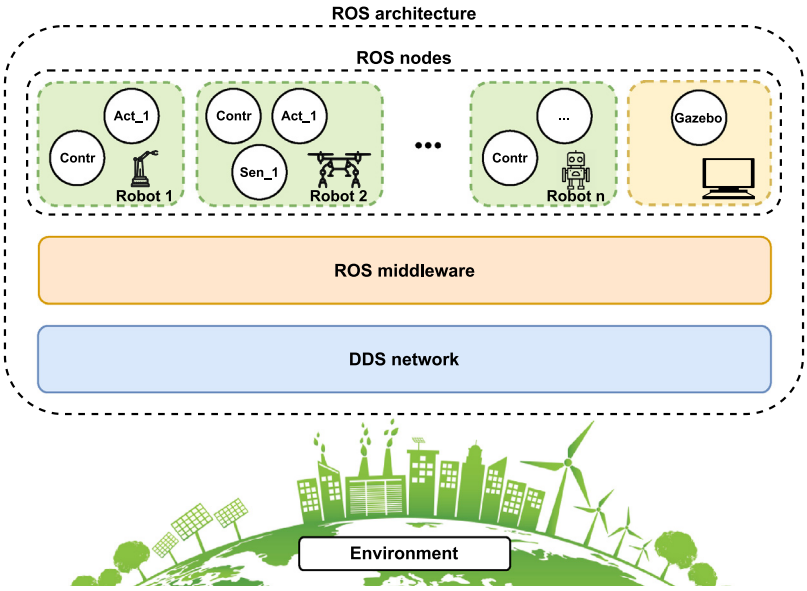


Fig. 1. ROS architecture.

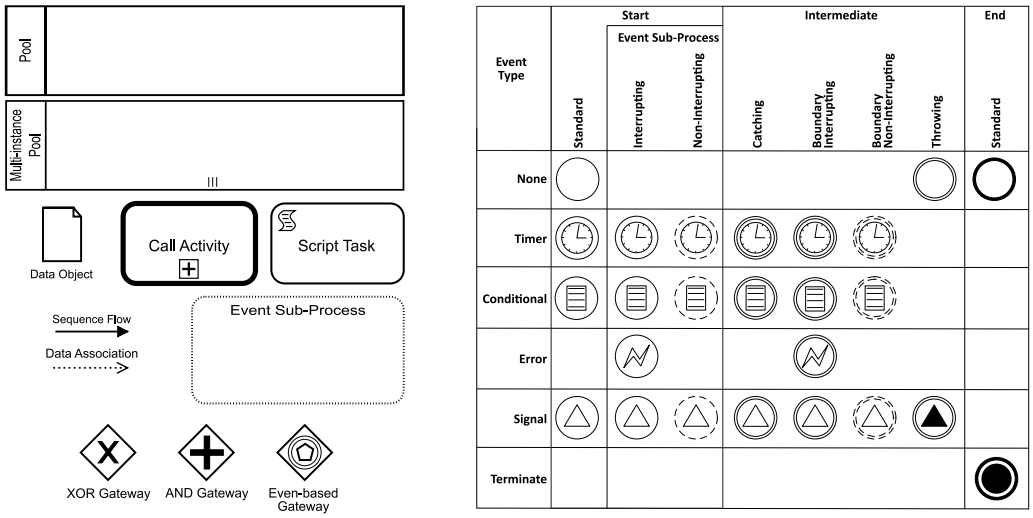


Fig. 2. Selected BPMN elements.

lower border represents multiple instances playing the same role. This means that a multi-instance pool represents a collection of entities sharing the same behavior. Only at run time, when the multi-instance pool is instantiated, its behavior is associated with a specific entity. To ensure that instances can be distinguished, the BPMN notation prescribes mechanisms based on communication [9, 8.3.2], such as message correlation or the use of an explicit identifier in the message payload.

In its turn, a *process* specifies all the possible behaviors an entity can perform, depending on the specific instantiation we are considering. It consists of process elements (e.g., activities, gateways, and events) connected through sequence flows and data objects linked to activities and events using data associations. Activities, rectangular boxes with round corners, represent one or more pieces of work to be performed within a process. Specifically, a *Call Activity* represents a reference to another process defined in a different BPMN diagram. This element enables structuring (possibly large and complex) models in terms of decoupled reusable processes. Graphically, it is characterized by a thicker border and by the *plus* symbol below the activity name. A *Script Task*, a rectangular box containing a *written document* symbol, consists in a piece of code to be executed at run-time. An *Event Sub-Process* is a kind of sub-process that acts as an event handler. Its graphical element is a rectangular box with a dashed border that can contain a process to trigger under certain conditions. The event sub-process is not part of the normal flow of its parent process but can be triggered concurrently by the firing of an event.

Other flow elements we consider for modeling MRSs are the *Events*, which are graphically depicted as circles with different inner symbols. They represent something that can happen at the beginning, during, or at the close of the process execution (*Start*, *Intermediate*, or *End* events, respectively). *Start* events are alternative entry points for enacting a process. When used in an event sub-process, they can interrupt or not the parent process (we refer to interrupting and non-interrupting events, respectively). *Intermediate* events can happen during the normal flow of the process when they are connected with sequence flows, or during an activity execution, when they are attached to the activity border. *End* events represent the possible termination of a process or a sub-process. All these events can be further characterized to describe their type and different semantics. Specifically, we

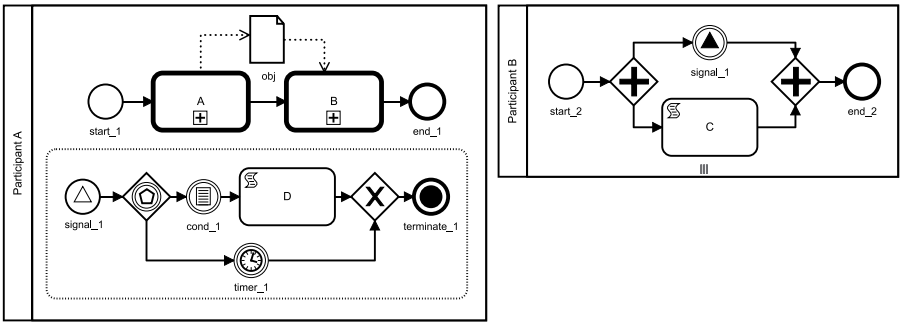


Fig. 3. Example of a BPMN collaboration diagram.

consider *Timer* and *Conditional* events, to react respectively to a time delay or a condition; *Error* events, to throw or catch execution errors; *Signal* events, to describe broadcast and point-to-point communications that may carry data (see [9, p. 235]); and the *Terminate* events to kill all the processes in a pool.

Gateways are used to manage the flow of a process both for parallel activities and choices. Gateways are drawn as diamonds and act as either join nodes (merging incoming sequence flows) or split nodes (forking into outgoing sequence flows). The *XOR* gateway represents an exclusive decision, the *AND* gateway a parallel execution, and the *Event-based* a choice based on events.

Finally, *Data objects* are information and material flowing in and out of activities and events.

Notably, we do not consider message flows in our BPMN fragment; in fact, all communications are modeled using signals, which better reflect the topic-based communication mechanism provided by DDS, which ROS adopts for intra- and inter-robot interactions.

The execution semantics of BPMN is token-based [9, Sec.7.1.1], somehow inspired by the semantics of Petri nets. Commonly, a token traverses, from a start event, the sequence edges of the process and passes through its elements enabling their execution, and an end event consumes it when it terminates. The distribution of tokens in the process elements is called marking, therefore the process execution is defined in terms of marking evolution. Referring to the collaboration diagram in Fig. 3, the example model contains a simple pool named *Participant A* on the left and a multi-instance pool named *Participant B* on the right. Its execution starts with a token in both the start events of pools *Participant A* and *Participant B*, i.e., *start_1* and *start_2*, respectively. Therefore, the token in *start_1* executes in sequence the call activities *A* and *B*, and ends once it reaches the end event *end_1*. Notice that activity *A* produces the data object *obj* that is taken as input by activity *B*. The token in *start_2* executes in parallel the throwing of *signal_1* and the script task *C*, and then its stops in *end_2*. Notably, once thrown, signal *signal_1* is cached by the catch signal with the same name in pool *Participant A*. This enacts the event sub-process which, in the case *cond_1* holds, executes activity *D*. Otherwise, once *timer_1* elapses, the execution passes directly to the terminate end event *terminate_1* that ends all the running executions of *Participant A*.

2.3. Running scenario

To better discuss how to specify the behavior of an MRS, we present here a running example depicting a smart agriculture scenario. The cooperation between unmanned ground vehicles, e.g. smart tractors or harvesting robots, and unmanned aerial vehicles, usually called drones, is a promising solution to achieve a fully autonomous and optimized farming system.

The proposed application scenario consists of two, or possibly more, tractors and one drone that cooperate to identify and remove weed grass in a farmland to enhance the yields. The drone and the tractors are equipped with a controller, enabling computations and communications, a battery, and several sensors and actuators. The drone is the only robot that can start its behavior at the system start-up: it receives the field's boundaries to inspect and starts the exploration. During the overflight of the area, the drone uses the camera to recognize weed grass areas and, when found, it sends to the tractors the coordinates. This triggers the tractors, which store the weed grass coordinates and send back to the drone their distance to the weed grass area. The drone can hence elect the closest tractor and notify it. At this point, the selected tractor starts moving towards the field, avoiding possible obstacles. Once it reaches the weed grass area, it activates the blade to cut the weed and stops its process until it receives a new position from the drone.

3. The FaMe framework

In this section, we present the FaMe framework we defined for modeling and executing robotic systems via BPMN collaborations. Fig. 4 depicts the framework and highlights the supported development phases: **modeling**, **configuration** and **enactment**.

The modeling phase corresponds to the first step in the development of a MRS using the FaMe framework. It consists of a disciplined use of the BPMN notation allowing the definition of a collaboration diagram that can represent at a high level of abstraction the behaviors of the robots involved in a MRS. The configuration phase enriches the modeled collaboration with all the information needed for its execution. It produces as output an executable collaboration diagram used in the following phase to command each robotic device, thus guaranteeing a decentralized execution of the MRS. The last phase is the execution of the BPMN collaboration directly on each involved robot, without the need for any direct translation into code. This is made possible through BPMN engines, one for each robot, that enact only the process associated with each considered device. As a result of this phase, the framework enables the MRS execution both in simulated and in real environments.

3.1. Modeling phase

We introduce here a list of guidelines to be applied during the modeling of a BPMN collaboration diagram using the selection of BPMN elements presented in Section 2.

The following guidelines define a disciplined use of the BPMN elements leaving the designer enough freedom to specify the MRS mission.

G1	**Robots as pools.** Robots involved in an MRS are abstracted by pools, representing the participants in the collaboration.
G1.1	**Heterogeneous robots as single-instance pools.** Robots that are heterogeneous, i.e., robots of a different kind or robots with different missions, are abstracted by single-instance pools.
G1.2	**Homogeneous robots as multi-instance pools.** Robots that are homogeneous, i.e., robots of the same kind with the same mission, are abstracted by multi-instance pools. This simplifies the resulting diagram, as a multi-instance pool represents many robots in terms of several instantiations of the same process, avoiding to repeat the same robot process into different pools.
G2	**Mission as a process.** The mission to be performed by a robot is abstracted by a process diagram within the pool of the considered robot. The process diagram expresses the control flow of the mission.
G2.1	**Actions as activities.** A robot mission is mainly made up of a set of actions; these actions are abstracted by activities within the mission process diagram. Based on the complexity of the action, activities in the model can be either call activities or script tasks.
G2.1.1	**Complex actions as call activities.** Complex actions, e.g., navigation, perception, and control, that can be decomposed into several steps and/or reuse already modeled procedures, are abstracted by call activities. Indeed, a call activity can be used for referencing another process diagram or other existing activities. This enables the modularization of diagrams and reduces their size, speeding up the modeling of the MRS through the re-use of already modeled behaviors.
G2.1.2	**Simple actions as script tasks.** Simple actions, which do not require any further decomposition, are abstracted by script tasks. This element refers to a piece of code[a] to be executed by the considered robot.
G2.2	**Event handlers as event sub-processes.** Procedures handling an event (such as the expiration of a timer, the satisfaction of a condition, the occurrence of an error, or the reception of a signal) are abstracted by event sub-processes. Indeed, this element triggers a handling process concurrent to the main process describing the robot mission. Based on the event type, the main process can be interrupted or not.
G2.3	**Concurrent behaviors by means of AND gateways.** Concurrent behaviors in a robot mission are rendered by means of AND split gateways.
G2.4	**Conditional choices as XOR gateways.** Conditional choices in a robot mission are rendered by XOR split gateways.
G2.5	**Event-based choices as event-based gateways.** Choices driven by events in a robot mission are abstracted by means of event-based gateways.
G2.6	**Missions activation as start events.** The beginning of a mission is abstracted by a start event. In more detail, a none start event fires immediately the robot mission. The other event types activate the mission when specific circumstances occur.
G2.7	**Missions shutdown as end events**. The end of a mission is abstracted by an end event. The none end event stops only the incoming execution flow, while the terminate end event stops any execution flow still active[b].
G3	**Communication via signal events.** Intra- and inter-robot communications, even in presence of a payload, are abstracted by signal events. The sending of a message corresponds to a throwing signal event, and the receiving of a message is modeled by a catching signal event. The correlation between one or more senders and one or more receivers is established by means of the event name.
G4	**Data as data objects.** The data used during the execution of the robot's mission are abstracted by data objects. They provide storage in which activities and signal events can read or write information. We explicitly represent in the model, in terms of data objects, only the information used to drive the decision-making and the data exchange in the mission execution. Of course, at the implementation level, other data will be required. However, since they are confined within low-level pieces of code and do not play any role at the abstraction level of the model, they are omitted. This permits reducing the complexity of the diagram.

[a]The code can be of any programming language supported by the robot's software.
[b]Notably, more than one execution flows in a model can be active due to an AND gateway (see G2.3).

Considering the running scenario of Section 2.3, the BPMN collaboration in Fig. 5 is a possible result of the application of the proposed guidelines (for sake of presentation, we relegate the BPMN models referenced by the call activities in Appendix A). As prescribed by guideline G1, we modeled the scenario using two pools: a single-instance pool for representing the drone (G1.1), and a multi-instance pool for representing the tractors (G1.2). In turn, these pools contain the robots' mission specifications in the form of processes (G2). For instance, the process contained in the drone pool depicts the sequence of activities that the robot has to perform, from the take-off to the landing. The other guidelines are then used in the modeling of each process which finally results in a composition of process elements (activities, gateways, and events) connected to each other via sequence flows. As prescribed by G2.1.1, we used call activities for representing complex actions that are specified in external BPMN diagrams, and that can be possibly reused several times in different parts of the collaboration. Differently, simple actions that cannot be further decomposed are rendered as script tasks (G2.1.2), for instance, the script task *Update Closest* performs simple mathematical operations to calculate the tractor closest to the weed. Finally, as indicated in G2.2, an event sub-processes is used for representing the procedure to handle a specific situation: *Weed Found* is an interrupting event sub-process that is performed whenever a weed is identified. Concerning the gateways, in the tractor's pool, an example of conditional choice is the decision taken by the tractor to determine whether or not to stop its execution, this is rendered as a XOR gateway with two possible outcomes (G2.4). Notably, the *"I'm the closest tractor?"* is the label of the XOR gateway, used to improve the readability of the model. As prescribed by the BPMN standard, the conditions of XOR gateways are provided as boolean expressions specified as attributes of the elements during the configuration phase. Whereas, an event-based choice happens when it waits for a *closest_tractor* signal. Indeed, if the signal is not received within 30 s, the timer event is triggered, routing the execution to another path in the model. This is rendered by means of an event-based gateway (G2.5) followed by two catching events: the signal and the timer. Regarding the events, guideline G2.6 leads us to use a none start event at the system start-up in the main process of the drone, while we use a signal start event for the tractors, which indeed are enacted only when

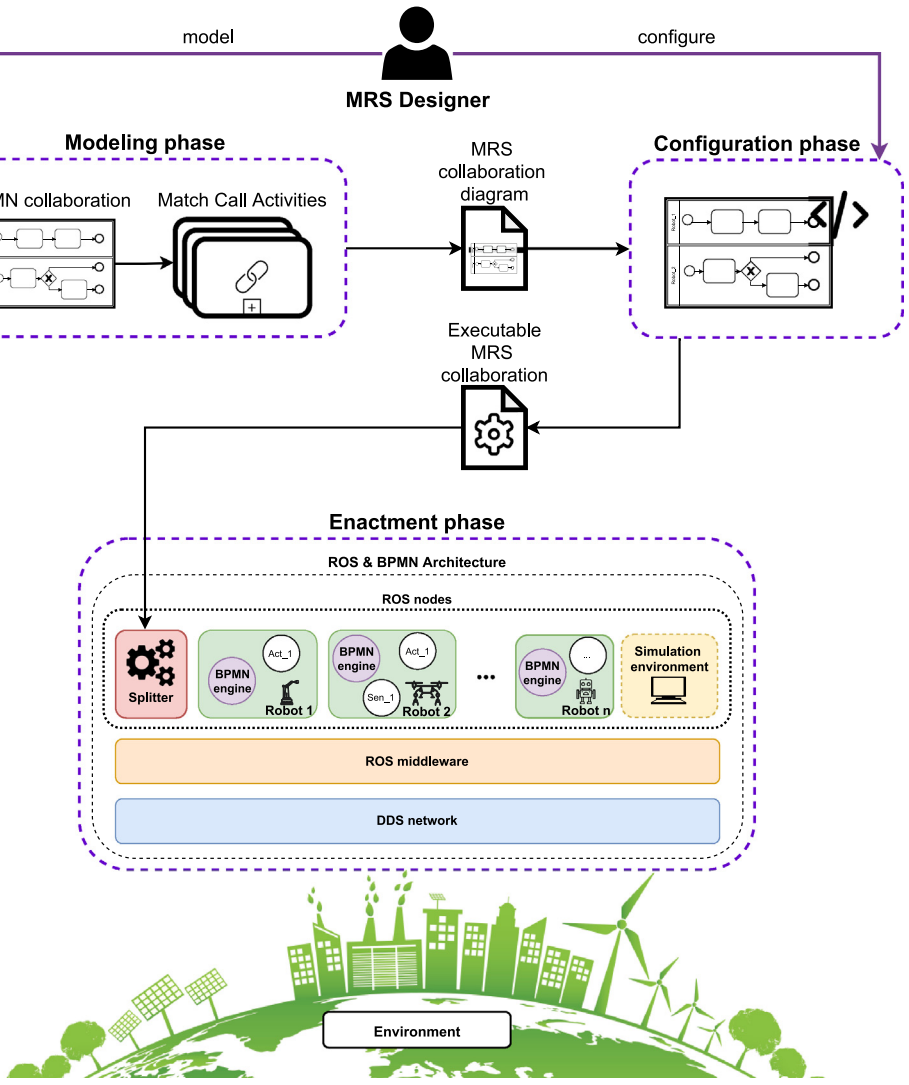


Fig. 4. The FAME framework.

they receive a *weed_position* signal. This happens when a signal with the same name is thrown, like for the signal event in the drone pool (G3). In conclusion, as prescribed in G4, the data used along the robot's process are represented through data objects, as for data object *Weed Position* which contains the coordinates of the grass to be removed.

3.2. Configuration phase

The collaboration diagrams we obtain using the provided guidelines can be executed through engines that implement the execution semantics of BPMN. Such engines reproduce the marking evolution of the model.

Referring to the collaboration we modeled for representing the running scenario (Fig. 5), its execution is as follows. At the system start-up, the only entry point that can be triggered is the none start event in the drone's pool, since the other start events are bounded to the triggering of events. Therefore, the drone starts its main process by performing immediately the *Take Off* call activity (see Fig. A.16), which refers to a process implementing all the activities concerning the lift-off of the robot. Once the take off has been performed, the next activity, *Explore*, refers again to the drone. This call activity enacts a process (see Fig. 6) that takes as input a data object containing the map of the area to be explored. In case the exploration does not reveal any weed to remove in the field, the control flow comes back to the main process of the drone, which terminates its duty (and the entire collaboration) going back to the base station by performing the *Return to Base* (see Fig. A.19) and the *Land* (see Fig. A.20) call activities. Otherwise, if the drone spots some weed grass, the *Explore* process triggers a signal event named *weed_found*. Consequently, the interrupting signal start event, contained in the event sub-process of the drone's pool, is triggered. Now, the sub-process blocks the *Explore* activity and performs its behavior. In detail, the sub-process triggers the sending of a signal named *weed_position* that contains the weed coordinates and starts the tractors' processes.

Indeed, the main process in the tractors' pool starts when it receives a *weed_position* signal. Then, each tractor performs the *Get Position* script task, sends back to the drone its position and waits either for a *closest_tractor* signal or for the passing of 30 s. In the meantime, the drone is waiting for the positions of the tractors, corresponding to the *tractor_position* catching event. In case no position arrives within 10 s, the process loops back to the throwing of *weed_position* signal, otherwise the drone updates accordingly the *Closest Tractor* data object. This is also the input for the next step in the drone's process execution, which assigns the duty of removing the weed to the closest tractor by triggering the *closest_tractor* signal. Thus, every tractor is informed of who

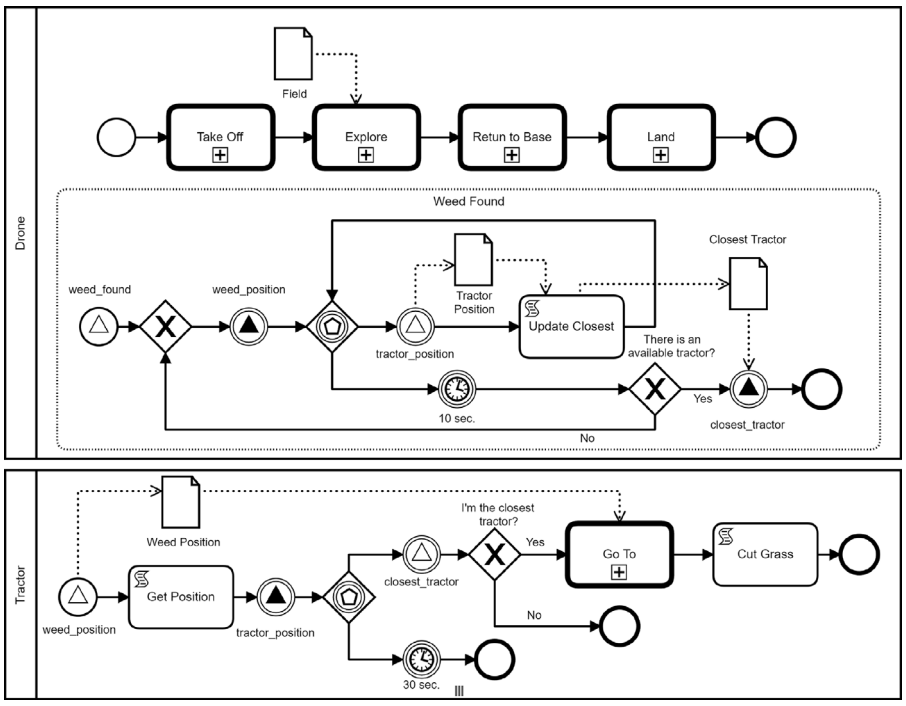


Fig. 5. Collaboration diagram of the smart agriculture scenario.

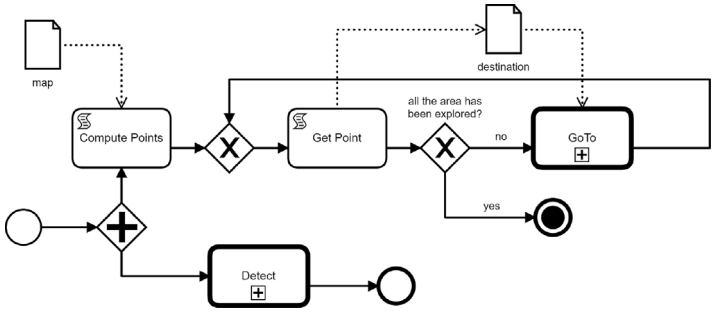


Fig. 6. *Explore* call activity.

is responsible for removing the grass, so that only this instance of tractor goes ahead to the *Go To* call activity, while the others terminate their process. The *Go To* call activity (see Fig. A.17), takes as input the weed position and performs the movements required to reach it, finally, the *Cut Grass* script task orders the robot to remove the grass in the reached position and the process terminates.

Notably, after the sending of the *closest_tractor* signal, the drone terminates the activities involved in the event sub-process, thus the control flow is given back to the main process (stuck in the exploration). This allows the drone to continue the field exploration and, in case other weeds are found, the collaboration described so far is repeated.

However, the described execution lacks a concrete link with the robotic ecosystem. To configure the BPMN collaboration obtained from the modeling phase to be executable directly by robots, the MRS designer has to provide in the BPMN file additional details about communication, scripts, and variables. These three aspects can be specified by inserting directly in the BPMN model pieces of code executable by the robots. We describe in the following the required configurations referring to the architectural aspects of ROS (Fig. 1). However, the modular architecture of the FAME framework permits replacing this ROS-based configuration phase with one tailored to other robotics languages.

The MRS designer has to act on the BPMN model by adding the required information in the *extension element* of the BPMN element under consideration. The extension element is part of the BPMN metamodel [9, 8.2.3] and is devoted to storing additional attributes while ensuring compliance with the standard. The element involved in the configuration phase are script tasks, signal events, and data objects, and are extended including the following information (for the sake of readability, the list of all configurations required in the running example are omitted here, but listed in Appendix B):

- **Data Objects**. As prescribed in guideline G4, data objects are containers for information used by the process elements. Therefore, the designer has to use such elements to instantiate the necessary variable names. Considering the running example, the data object *Closest Tractor*, depicted in Fig. 7, contains the declaration of variable `${closest_tractor}` whose value is dynamically updated by the script task *Update Closest*. Indeed, the other process elements can refer to such value using the variable name.
- **Signal events**. In ROS, the communication, i.e., the publication/subscription of messages over a topic, is specified using: a *topic name*, a *type*, and a *payload*. Concerning the topic name, it is inferred looking at the signal event name

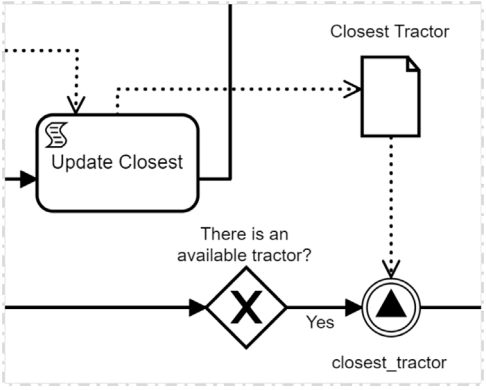


Fig. 7. *Closest Tractor* data object in the main process.

defined by the designer during the modeling phase. While the types of the message (e.g., string, float, bool) and its payload need to be added manually. For instance, in Fig. 5, the signal event *closest_tractor* has to be decorated with the type `String` and the payload `${closest_tractor}` indicating a message of type string and a payload containing the name of the closest tractor, respectively.

- **Script tasks**. Depending on the desired outcome, the MRS designer has to include in each script task the lines of code to be executed at run-time by the corresponding robot. Considering again the running example in Fig. 5, the script task *Update Closest* (Listing 2 in Appendix B) calculates which tractor is closest to the detected weed. Therefore, the task contains the code that makes the robot perform this computation.

Once these configurations have been done, the BPMN collaboration can be deployed and the MRS executed.

3.3. Enactment phase

The enactment phase consists of the deployment of the BPMN collaboration in the robotic system and of its execution on real or simulated systems.

The first step in the deployment is the splitting of the BPMN collaboration into several process models, one for each robot. This is because the framework is defined to realize a distributed and decentralized execution of the MRS. The splitter component extracts the pools from the collaboration diagram and associates each of them with the corresponding robot. Notably, to bind a pool to a robot, we use the same name for the robot namespace in ROS and for the pool in the diagram. In the case of a multiple instance pool, the splitter component searches for namespaces starting with the pool name, followed by the symbol "_", and ending with some digits. This permits automatically determining the number of robots corresponding to the multi-instance pool, in this way the same model can be used in scenarios with a different number of robots.

Following the distributed architecture of ROS, where a robot is a collection of nodes, we included in the robots a node implementing a BPMN engine. Each of these nodes receives the pool to execute from the splitter component and recreates a BPMN execution platform on top of the robotic environment. Thus, the engines enable the robots to perform computations and communicate with each other. Since the splitter deploys every single process separately, the designer can change at run-time the behavior of all robots or of a subset of them, even of those that have been previously instantiated with a multi-instance pool. The enactment of the modeled collaboration can be done in the same way in real scenarios or in the Gazebo simulator, thanks to the ROS infrastructure which considers Gazebo as a node in the DDS network. Therefore, the MRS developer can decide to execute the collaboration directly on the real robots or simulate the scenario in order to spot potential issues.

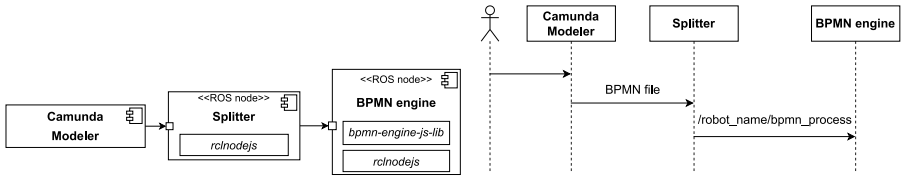
At the startup of the MRS system, each engine fires the execution of its part of the BPMN collaboration following the BPMN execution semantics. Every time an engine triggers a BPMN element that involves the performing of an action by the robot (i.e., a script task or a signal event), the code embedded in the element is evaluated at run-time by the robotic framework. The other elements instead are entirely interpreted by the engine; for example, when an engine executes a XOR split gateway, it drives the control flow towards the sequence flow with the true condition without sending any command to the ROS framework. In the case a process or an activity is interrupted during its action, like when a terminate end event is fired or when we want to deploy a new process at run-time, the engine forces the termination of any action the robot is performing. This, of course, may provoke risky situations; for instance, a drone may fall during its flight, or a tractor can continue to go ahead indefinitely. We handle these situations directly through the implementation of the activities by bringing the robot to a safe state (i.e., the drone lands and the tractors stop) before the engine kills the process or the activity.

4. The FAME toolchain

The FAME framework is supported by a toolchain composed of already existing software and artifacts we developed for experimenting with the model-driven development approach we propose for ROS2-based MRSs. Fig. 8(a) provides a component diagram to better clarify how the software components of the FAME toolchain are connected. The modeling, configuration, and execution phases can be *fully* automated through the use of our toolchain. Source and binary code of our software tools included in the toolchain, as well as a user guide and an *how-to* video guide, are available at the following link: https://pros.unicam.it/fame.

The modeling and the configuration phases described in Sections 3.1 and 3.2 produce models compliant with the BPMN standard notation. Therefore, any standard BPMN modeler can be used in these phases among the many that are already available (we refer to https://bpmnmatrix.github.io/ for a non-exhaustive list). We have chosen the Camunda modeler (https://camunda.com/download/modeler/) since it is cross-platform, free, and easy to use.

The automation of the execution phase described in Section 3.3 is supported by our implementation of the splitter and of the BPMN engines as ROS nodes (see Fig. 4). The splitter node implements a procedure for extracting from the input BPMN collaboration the involved processes and sending them in the form



(a) Component diagram (b) Sequence diagram

Fig. 8. The FaMe toolchain.

of a string over ROS topics named */robot_name/bpmn_process*. The ordering, as well as the interaction and the exchange of data between the presented tools, is depicted in Fig. 8(b). Specifically, the BPMN engine node is a customized version of the JavaScript BPMN engine (https://github.com/paed01/bpmn-engine). We integrated it into the ROS architecture, making possible the interpretation of BPMN models configured as prescribed in our approach. The integration of the engine has been done without any limitation as we exploit the *rclnodejs* client library (https://github.com/RobotWebTools/rclnodejs) for interpreting nodes written in JavaScript. Anyway, we extended some of the engine features to support the FaMe framework as follows. The initialization of the engine creates a scope of variables, named *environment*, containing the model and the ROS engine node itself. Moreover, the activation of a data object adds the data it stores in the environment. As already discussed in Section 3.1, the communication is managed by signals mapped into ROS topics. To tailor the engine functionalities with the ROS communication model, enacting a throw signal creates a ROS publisher characterized by the information retrieved from the model, and a catch signal produces a subscription to the related ROS topic. The message passing over ROS topics allows communication among signals executed in different engines by producing the termination of the catch signal as soon as the subscriber receives a message. We provide a FaMe ROS package (https://bitbucket.org/proslabteam/fame/src/master/fame_engine) containing the splitter node and the engine node, thus allowing its reuse and integration with different robots.

5. Experiments

This section presents the result of the experiments we conducted to assess the feasibility of the FaMe framework in practice. The maturity of FaMe has been evaluated by means of concrete execution of two MRSs: one regards the running example introduced in Section 2.3, while the other regards a *Ground Vehicles Cooperation* scenario we introduce later in this section. The two scenarios have been executed in the simulation environment offered by Gazebo. The latter has been also reproduced using real robots acting in a real-life environment.[1] The choice of proposing two examples was driven by the aim of showing the application and the potential of the FaMe framework both in a complex system belonging to a specific application domain and involving heterogeneous robots, i.e., the running example, and in a simpler scenario that can be easily developed with physical robots, i.e., the ground vehicles cooperation.

We provide all material needed to reproduce the experiments. Indeed, all the artifacts we show in this section, the source code of the examples, instructions, video tutorials, and references are made available at the following link: https://pros.unicam.it/fame.

[1] The running scenario, instead, has not been executed in the real world, since we did not have drones, tractors and an agriculture environment suitable for the experiment.

5.1. Experiments setup

We introduce below how to setup up the robotic environment, and the FaMe framework, to execute the experiments.

For the MRS, both in the physical and simulated environment, it is necessary to install the ROS framework, the required libraries, and dependencies on the involved devices. Each deployed robot is composed of one node for each sensor, one node for each actuator, and the BPMN engine node to control the behavior. When developing physical robots, the nodes related to onboard sensors and actuators must be designed so that ROS is able to obtain low-level information from the sensors or send commands to the actuators. Differently, in simulated experiments, 3D models are needed for representing both the robots' components and the environment, such as the tractors' wheels or the weed grass.

To achieve a correct system operation, the designer needs to properly follow the configuration phase, and satisfy the following constraints. The name of each pool must be equal to the namespace associated with the related robot. In this way, the splitter can identify the robots participating in the ROS network and share the corresponding BPMN process. The identifier of a topic, i.e., the signal name, must match the corresponding topic name that is configured in the robot.

5.2. Simulation-based experiment

Here we describe the experiment we carried out in the simulated environment on the smart agriculture running example (the other experiment can be similarly reproduced using the code and the instructions we provide in the FaMe website).

After completing the modeling and configuration phases, the MRS designer can execute the simulation of the scenario. To this aim, the designer has to launch all the nodes composing the system: the splitter, the engine nodes, and the node related to the Gazebo simulation. As a result, the MRS designer can take advantage of the graphical interface of Gazebo that, in this case, will show graphics representing the drone and the tractors. The robots start performing their activities according to the behaviors modeled in the BPMN collaboration. Fig. 9 shows two stages of the MRS collaboration, developed using the FaMe framework. The objects in red are the tractors, while the white one is the drone. Notably, the blue area projected from each robot corresponds to the view range of the ultrasonic sensor. In more detail, Fig. 9(a) depicts the drone performing the exploration of the field and detecting a weed grass, while Fig. 9(b) shows the moment in which the drone triggers the tractors and assigns the duty of removing the grass to the tractor closest to the weed.

5.3. Real-life experiment

For the real-life experiment, we consider a scenario that involves two cooperative ground vehicles with different capabilities, but with a common mission, i.e. identify a specific target and destroy it. REX (Robot EXplorer) is in charge of performing the exploration of the area and identifying the target that

(a) Weed identification (b) Weed removal

Fig. 9. Running scenario simulation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

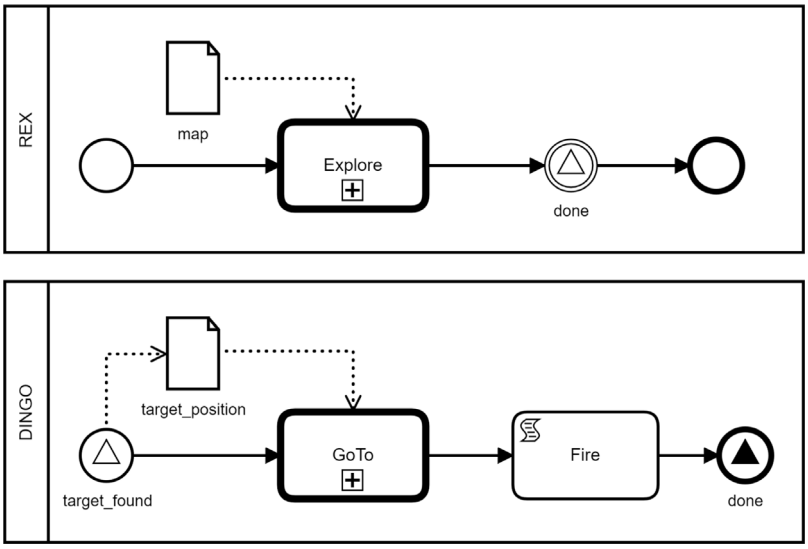


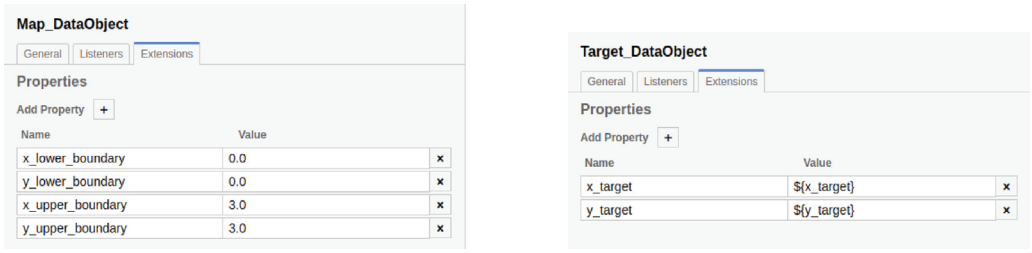
Fig. 10. Ground vehicles cooperation.

should be destroyed. Whereas, DINGO (DestroyING rObot) starts its execution when it receives the coordinates of the target, so that it is able to reach and destroy it. The destruction of the target determines the ending of the system execution. Below, we analyze the application of the FAME framework to this case study.

The obtained BPMN collaboration model is depicted in Fig. 10. At the system start-up, REX is the only one that starts its execution by performing the *Explore* call activity, to detect the desired target. When the exploration process ends, the robot moves to an idle state waiting for a *done* signal, before ending its main process. In case the exploration spots the target, this activates the DINGO's process, via the *target_found* start signal. Therefore, DINGO performs the *GoTo* call activity to reach the target and destroys it by enacting the *Fire* script task. Its process ends with the throwing of a *done* signal that determines the end of the system's mission. The model is enriched with all the information required by the enactment phase. Therefore the *map* data object, as shown in Fig. 11(a), is configured to store four values for the x and y axes representing the boundaries of the area of interest, whereas, the *target_position* data object contains values that depends on the payload of the caught signal (Fig. 11(b)), i.e., the position of the target on the x-axis and on the y-axis. The *done* signal is extended to be compatible with a boolean message type (Fig. 12). Therefore, the catching signal only stores that the signal can handle boolean data, whereas the throwing also sets up the carried payload, equal to `true`.

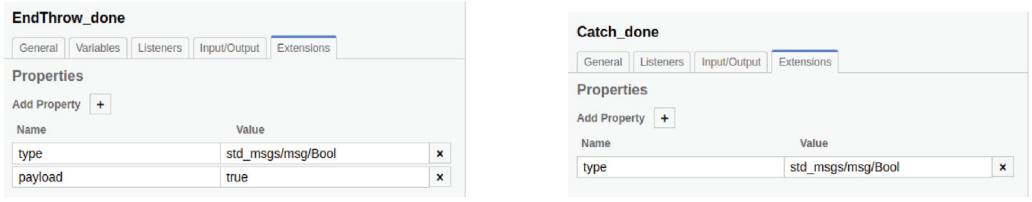
Finally, the *Fire* script task is configured to send an internal command to the robot, to manage a specific actuator. Notably, we decided to implement this action with the activation of a red LED. The obtained code, reported in Listing 1, initializes a publisher in the BPMN engine node able to send a `ColorRGBA` message type over the `led` topic. This publisher is then in charge of publishing a message equal to the red color.

The enactment of the BPMN collaboration is executed firstly in the simulated environment to visualize the system behavior before moving on to the real-life experiment. In order to execute the experiment in the real world, we exploited two smart cars equipped with four wheels, an ultrasonic sensor, an RGB LED module, and a Raspberry Pi 3B+ as onboard computational unit. We deployed in the robots the ROS packages needed to control their devices and the FAME package that runs the BPMN engine node. Therefore, once the BPMN collaboration model is processed and executed, the cars were able to perform their collaborative behavior. Fig. 13 shows part of the experiment; we put in comparison the simulated and the real-world executions to emphasize their one-to-one correspondence. Accordingly to the models, we can see that *REX* starts its execution with an exploration of the area (Fig. 13(a)) and is able to identify a target (Fig. 13(b)), which activates the *DINGO*'s process, that reaches the target (Fig. 13(c)) and destroys it (Fig. 13(d)). All the phases of the FAME framework applied to this scenario are available step-by-step in the video tutorial in the FAME website.



(a) Map configuration (b) Target configuration

Fig. 11. Data objects configuration.



(a) *Done* throwing configuration (b) *Done* catching configuration

Fig. 12. Signals configuration.

```
const node = this.environment.variables.ros_node;
const publisher = node.createPublisher("std_msgs/msg/
    ColorRGBA", "led");
publisher.publish({r: 1.0, g: 0.0, b: 0.0, a: 0.0});
next();
```

Listing 1: *Fire* script task

5.4. Performance evaluation

We now show the result of the experiments we made to assess the performances of the model interpretation approach of FAME. We measured the use of system resources and the impact of possible delays related to the model interpretation. The experiments have been conducted on a virtual machine based on the Ubuntu 20.04 operating system. The system has 8 GB of RAM, a quad-core CPU, and all the tools and dependencies required by FAME installed.

Referring to the use of system resources, we measured the CPU and memory usage on ten executions of the real-life experiment shown in Section 5.3 both by directly compiling the source code and by interpreting the model with FAME. The consumption of the system resources has been recorded using the `top` command.

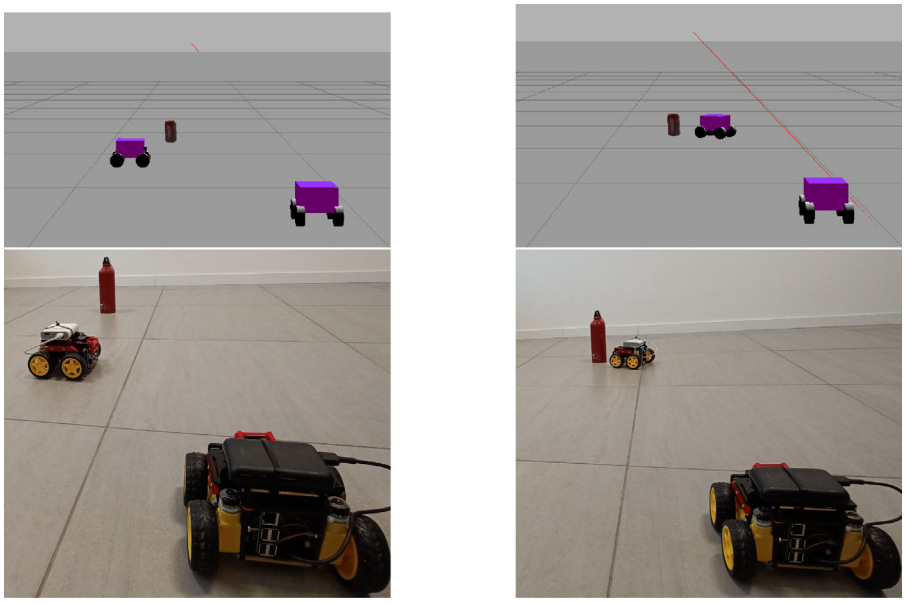
Fig. 14(a) shows the total CPU time used by the controller task, expressed in hundredths of a second. In the direct code execution, this amount of time is lower than in the model interpretation approach. Fig. 14(b) expresses the amount of memory that may be simultaneously accessed by multiple programs, expressed in megabytes. Here, the model interpretation performs more fluctuating and slightly worse behavior with respect to the code compilation. Fig. 14(c) plots the total amount of virtual memory used by the ROS nodes, including all the code, the data, and the used libraries, expressed in megabytes. In this case, the approaches produce very similar results, minimizing the difference between the two. Finally, Fig. 14(d) highlights how much memory the ROS nodes are consuming in the physical RAM, expressed in megabytes. The compiled code performs better than the interpretation of the model although, also in this case, the difference is minimal.

These results do not show a marked difference between the two approaches, especially if we consider that the ROS itself runs on powerful hardware. Therefore, we present below the result of the experiments conducted on a physical device, where we measured possible delays introduced by FAME. To conduct this evaluation, we exploited the robot presented in Section 5.3. Specifically, the robot is powered by a RaspberryPi 3B+ with 1 GB of RAM, a quad-core CPU, and the FAME dependencies installed. The experiment consists in making the robot move straight forward at a constant velocity in the direction of an obstacle, once it perceives a distance to the obstacle lower than 50 centimeters, it stops the wheels. Thus, we measured the distance between the obstacle and the point where the robot halted. The experiment has been conducted both with a direct code compilation and by exploiting the FAME framework and considering both a simulated and a real environment for the execution. We ran 10 times the experiment using three different speeds: 0.8 m/s, 0.9 m/s and 1 m/s. Fig. 15 summarizes and compares, on average, the distances perceived during the experiment. Specifically, a higher value corresponds to a better performance of the system, as it shows that the communication between the robot controller and the wheels has been performed quickly. Both in the simulated and in the real-life experiments the results are comparable, indeed there is not an approach performing better than the other. If we aggregate the results obtained at different speeds, on average, in the real-life scenario FAME stops the robot at 39.07 cm from the obstacle, 1.11 cm after the approach with compiled code that stops the robot at 40.18 cm. While, in the simulated scenario, FAME stops the robot at 14.09 cm while the approach with compiled code at 14.67 cm, with a difference of 0.58 cm. Notably, in the simulated environment, the final distance to the obstacle is slightly lower than in the real environment since the friction between the wheels and the ground is lower.

Summing up, the difference between the two approaches is not excessive, especially if we consider that FAME has not been meant to run MRSs operating in critical situations.

6. Related works

In the literature, many works face the model-driven development of robotic systems considering the different stages that



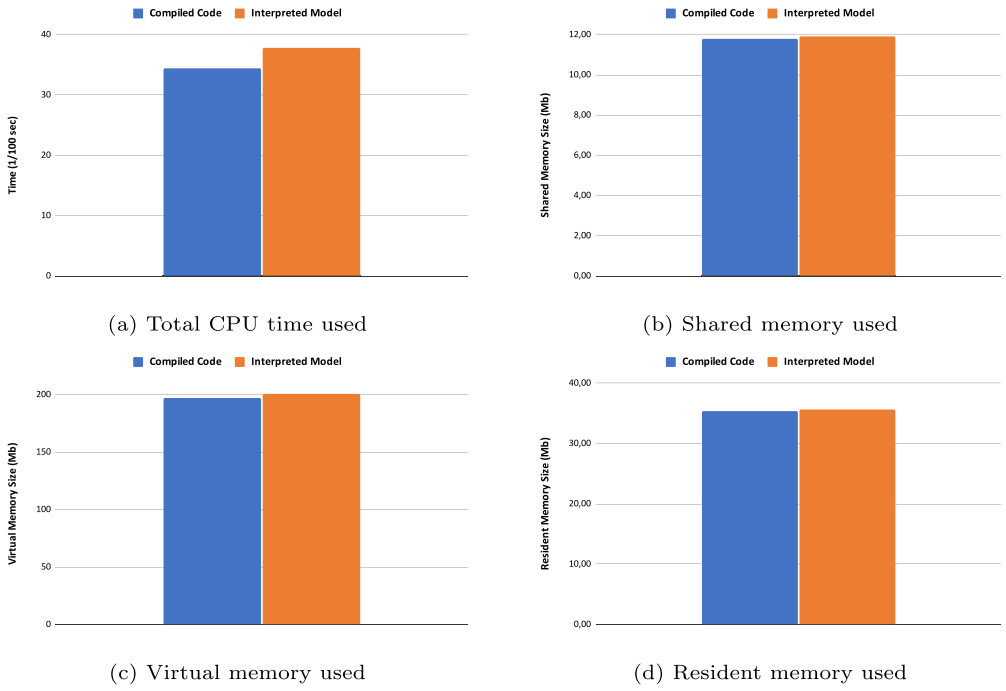
(a) REX exploration (b) Target identified

(c) DINGO activation (d) Destruction

Fig. 13. Real-life experiment.

lead from systems modeling to execution in real contexts. In this regard, these works show different levels of maturity and provide contributions mainly focused on: (i) modeling robotics missions, and enacting the obtained models via a central control unit, (ii) deploying and enacting the models inside the robots, and (iii) developing model-to-code translations leading to the execution.

Considering the approaches focused on the model execution managed by a central control unit, in [11] the authors describe a tool for the mission specification for a team of multicopters and the generation of a detailed flight plan, using a Domain Specific Language (DSL) that is translated into an intermediate language representing the basic actions of an aerial vehicle. These actions are combined to define the drones' mission using a finite state transition system where each transition corresponds to an action. Exploiting the same tool, in [12] the authors define a set of DSLs to specify the civilian missions for unmanned aerial vehicles. The mission specification and enactment are done through a web-based graphical interface, that communicates with some ROS-based controllers sending commands to the vehicles. Notably, using custom DSLs may require time to learn them. This effort could be mitigated using a standard modeling language, like the BPMN notation used in FAME. Indeed, BPMN is a well-accepted standard applied to fit a wide variety of process modeling purposes [13], thus resulting familiar to many users. Its notation is easy to understand and learn [14], especially considering that in FAME we identify and use only a subset of elements. On the other hand, the guidelines we defined to support the application of BPMN for modeling the mission of an MRS shift BPMN from a general purpose notation to a domain-specific one. Even if this means that our guidelines should be learned by the designers, the effort to learn them is not high. Indeed, they do not introduce any new element to the notation and do not



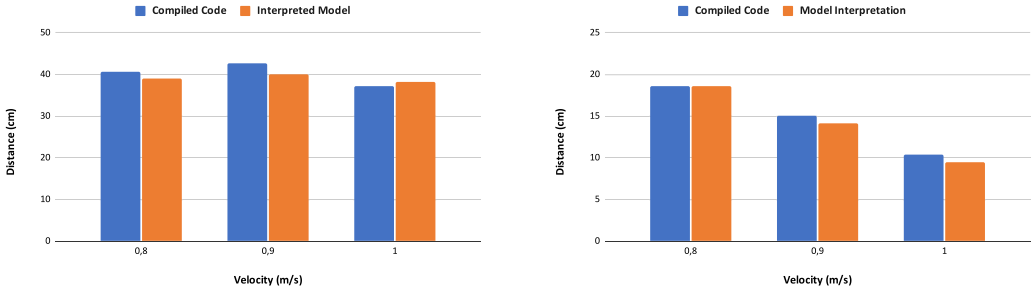
(a) Total CPU time used

(b) Shared memory used

(c) Virtual memory used

(d) Resident memory used

Fig. 14. Resources consumption measurements.



(a) Real-life experiment

(b) Simulated experiment

Fig. 15. Performances evaluation.

associate different semantics with the BPMN elements. The guidelines just provide disciplined arrangements of BPMN elements, linking them to the corresponding MRS concepts. In [15] the authors propose a laboratory survey to enable the co-creation of DSL solutions for robots. It aims to reduce the complexity and the necessary technological background that is required to develop a robotic application. The proposed tool running externally from the robot can use BPMN models to abstract the implementation of one single robot, and to generate and execute the related mission. Another integration of the BPMN standard with an autonomous robot is presented in [16]. The authors develop and automatize a warehouse process by implementing a human–robot cooperation system managed by a web interface able to control the entire process in real time. These approaches exploit an external device to enact the process and send commands to the robot, thus making this central device a single point of failure for the robotic system. The FAME approach, instead, is fully distributed; hence, it is more suitable in a multi-robot context. Nevertheless, in cooperative scenarios, the behavior of the robots can be interrelated. Thus, it may happen that if a robot fails, the entire mission cannot be carried on. In those cases, the MRS designer should prevent this situation by modeling an alternative behavior that reacts to a robot failure.

Moving on to the interpretation of the model executed directly by a robot, the authors in [17] describe the TRACE tool to tailor BPMN to the robotics domain in order to model a sequence of robotic activities. The aim of this approach is to understand what will happen after an unplanned event and check if it will compromise the mission. The authors applied their proposal to a single robot equipped with ROS1, which can execute its tasks and autonomously react to unexpected events. In the same direction, in [18], the authors present an application of the TRACE tool to a multi-robot scenario. The mission is specified in a BPMN file, uploaded inside all the vehicles, and each block of the model corresponds to a robot's behavior. The system uses the ROS1 framework with an open-source package that allows a custom message-passing among multiple master nodes. Such message passing creates a decentralized communication among ROS nodes implementing an ad-hoc communication mechanism that, differently from our approach, lacks to exploit the distribution brought by the powerful DDS standard. Moreover, since the robots share the same mission, but cannot communicate with each other, the authors highlight that a human operator must be involved in

assigning robots to different areas. Exploiting the FaMe tool, it would be easier to add an automatic coordinator to manage area coverage. At the same time, robots could handle this problem independently by leveraging distributed DDS communication.

Other approaches develop model-to-code translations for system execution. In [19] the authors provide an automatic method to verify the task-level control in autonomous robotic systems using Petri Nets as input language for a model-checking tool. A DSL is used to define the robot's behavior and all the constraints that must be satisfied. Each global action is then translated into a Petri Net and verified in order to check system fairness and efficiency. The approach uses a single-vehicle scenario, equipped with ROS1. In [20] the authors present a methodology for the description of a multi-robot mission and the related application to a ROS-based system, using Hierarchical Petri Nets. The proposal is extended in [21], in which a 6-layer Petri Net is used to describe the activity of the robotic system at a different level of abstraction. The resulting model is used to verify the system safety and liveness, and finally to generate ROS-executable code for controllers. In [22] the authors present a model-driven approach to support the systematic engineering of MRSs thus facilitating the definition, implementation, and analysis of these systems. The proposed framework is based on the ATLAS DSL, for the specification of the team structure and mission objectives, and on a code generator engine. The latter enables communication and coordination among system components, creates an interface for the direct interaction with a robotic simulator, and produces the related configuration files. Finally, in [23] the authors propose a platform to develop multiple robots by means of a user-friendly model editor. The editor is based on ROS1 so that users can graphically represent the system mission that is automatically translated into an executable Python source file. The works described above are based on translations of models into code, which limits dynamic changes to the mission of a MRS. On the other hand, our approach relies on the interpretation of models by a BPMN engine, which is more suitable to support change at run-time to the MRS mission.

The analysis of the literature review has been summarized in Table 1. The comparison shows that all the approaches exploit version 1 of ROS, therefore the communication is managed by a master node. This creates a centralized architecture that is unsuitable for the development of MRS applications. Indeed, many of these works do not consider MRSs at all. Furthermore, many approaches execute the system by exploiting the model-to-code translation instead of the execution of the model. This implies that the mission of the robots has to be uploaded offline and any modification of the model leads to the need to interrupt the system and load the new code generated by the translation. All these shortcomings are taken into account in the design of the FaMe framework. Specifically, distributed communication among robots is achieved with the adoption of the ROS2 middleware, and a direct enactment of the model is obtained with the deployment of a BPMN engine inside each robot.

Table 1
Features comparison of the related works.

Paper	Purpose[a]	Modeling language	ROS version	#Robot	Architecture
[11]	M	MML	ROS1	Multi	Centralized
[12]	M, E	MML	ROS1	Multi	Centralized
[15]	M, E	BPMN	ROS1	Single	Centralized
[16]	M, E	BPMN	ROS1	Single	Centralized
[17]	M	BPMN	ROS1	Single	Centralized
[18]	M, E	BPMN	ROS1	Multi	Decentralized
[19]	M, E	TDL	ROS1	Single	Centralized
[20] [21]	M, E	Petri Net	ROS1	Multi	Centralized
[22]	M, E	ATLAS	ROS1	Multi	Centralized
[23]	M, E	Domain-specific	ROS1	Single	Centralized
Ours	**M, E**	**BPMN**	**ROS2**	**Multi**	**Distributed**

[a]M: Modeling E: Execution.

7. Concluding remarks

This paper proposes FaMe, a framework that permits the exploitation of a disciplined use of the BPMN notation for modeling and executing MRSs whose software is based on ROS2 and DDS. In this regard, we identified a selection of BPMN elements and we defined guidelines driving the modeling of MRS missions via BPMN collaboration diagrams. The subset of elements we selected results expressive enough to represent different MRSs involving different kinds of robots conveniently. Besides modeling, the FaMe framework entails a configuration phase we defined for enriching the collaboration models with information related to the MRS execution. Indeed, the last phase of FaMe defines how to execute BPMN models to control real robots. The framework has been evaluated through experiments in simulated and realistic environments exploiting a toolchain that automates the modeling, configuration, and execution of MRSs.

7.1. Discussion

The definition of the framework results from a balance of opposing forces. On the one hand, we aim to provide models at a high level of abstraction to facilitate the vision of the cooperative MRS behavior. For example, we mastered the complexity of a large model in a standard way via modularization, i.e., the identification of *reusable behaviors*, like for the *Go To* activity of the running scenario, which indeed is used both in the tractor process and in the definition of the activities *Return to Base* and *Explore*. On the other hand, we were also driven by practical aspects of MRSs programming differently from other similar proposals. In this regard, the choice of targeting ROS2, and its DDS implementation of the communication system, is reflected in identifying the BPMN concepts necessary for modeling specific robotic aspects, like the publish/subscribe communication protocol.

Another strength of FaMe lies in using the BPMN collaboration diagrams as they are defined in the specification document of the BPMN 2.0 standard. Indeed, it is worth noticing that many of the works that foster the use of BPMN for modeling IoT and CPS (see [24] for a survey on this topic) propose extensions of the notation to capture domain-specific features. Differently, we show that it is possible to successfully model the behavior of a cooperative MRS without extending the standard. This brings two main advantages: (i) model designers are not required to learn new concepts, and (ii) existing BPMN tools can be used without any customization. In particular, the use of BPMN alleviates the burden on the shoulders of the designer for what concerns dealing with the technicalities required by the use of native languages for robotics. Moreover, we can rely on the existing techniques which support the analysis and verification of the control flow, like the ones from the Business Process Management community that support the development of high-quality systems, such as formalization [10], animation [25], and verification [26] approaches. This permits the detection and repair of issues related to the execution of the processes and their interactions directly on the MRS collaboration and before its deployment in the system. Nevertheless, even if the MRS collabo-

ration is correct, the configuration phase and the enactment could introduce further problems leading to unwanted outcomes, such as a deadlock. Indeed, during the configuration phase, the MRS designer could introduce a wrong piece of code that blocks the process execution. Instead, during the system execution, we could face unpredictable situations, related to the robotic hardware or to the environment, that are not managed in the collaboration model. For instance, a laser sensor could stop sending data, or mud could block the robot's wheels. In this sense, it would be useful to define approaches to guarantee the specification-level properties of the system in execution.

To better explain the reasons behind the choice of the model interpretation approach, we discuss below the differences with respect to the model-to-code approach. In a model-to-code solution, the modeling language is translated into a mainstream programming language. This solution may be convenient when the modeling language is almost a more abstract version of a programming language, while it results less suitable when the model elements have no simple counterparts in the target programming language. For example, simultaneous executions, system-level signals, and time constraints are common features in MRSs and can be challenging to translate, which may lead to generated code behaving in a way not compliant with the semantics of the source model. Moreover, the model translation does not allow the update of the model at run-time, thus any changes to the model will reset the connection to the related system [27]. In the model interpretation approach, instead, an execution engine is in charge of directly reading and executing the model. Therefore, this approach allows faster changes, since any change in the model does not require explicit regeneration and rebuild steps. Additionally, these changes can be updated at run-time, without stopping the running application, thus making this approach more suitable for autonomous systems.

In both approaches, we have to assume that the generated code and the interpreted behavior are correct with respect to the semantics of the model. Considering the model-to-code approach, the translator is usually created ad-hoc, so it could be prone to implementation errors. Whereas in the model interpretation, the designer can rely on a direct interpretation of the model via already existing engines, which are tested, used, and maintained by the community. This makes the interpreted behavior executed by the engines reasonably trustworthy.

It is also worth noting that in the model interpretation approach applied to an autonomous MRS, the interpreter should be installed and run inside each robot. To assess the impact of the interpreter on the robot system performance, we carried out experiments (illustrated in Section 5.3) to compare the performances in case of model interpretation with those in case of compiled code execution. The experiments show that, even if the model interpretation approach performs slightly worse than the code execution, its impact on the system resources load and on the whole application efficiency is not particularly significant.

7.2. Future work

The proposed framework paves the way for several improvements to address other aspects and challenges of the model-driven development of MRSs. Concerning the modeling, our main objective was to allow the high-level specification of each aspect of a MRS. Indeed, the designer can use BPMN elements like data objects and signals to express how data and control flows are related. However, in the configuration phase, the designer has to concretely specify these data using types, variables, and values. In future work, we aim at providing an extension to the standard BPMN modeler that automatizes, even partially, the configuration phase, for instance by giving to the designer the list of pre-defined options for setting the lower-level data. We also plan to extend the framework with a repository collecting BPMN models to be referenced by call activities. In this way, we can further facilitate the MRS designer, who can rely on ready-to-use behaviors and thus compose the collaboration diagrams by reusing *building blocks* [28].

Another aspect we plan to investigate regards the enactment phase and how FaMe handles run-time modification of the executed processes. Indeed, we intend to define a resilient mechanism for handling transient behaviors and avoid system failures related to the interruption of ongoing activities. Moreover, we intend to enrich the framework by introducing an analysis phase for spotting anomalies and security issues in the modeled collaboration diagrams using observations of the MRSs execution. In both cases, the literature already provides techniques suitable for the integration in FaMe. Indeed, verification of behavioral and security properties on BPMN models is already studied by the community [26,29], and process mining [30] results in a valid approach for the analysis of logs coming from systems' execution.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix A. Call activities

See Figs. A.16–A.21.

Appendix B. Configurations

B.1. Script tasks implementation

```
const env = this.environment.variables;
const x_weed = env.x_target;
const y_weed = env.y_target;
if (env.closest_tractor == null) {
    env.x_closest = env.x_tractor;
    env.y_closest = env.y_tractor;
    env.closest_tractor = env.tractor_name;
} else {
    var d_tractor = Math.sqrt(Math.pow((env.x_tractor - x_weed),
     2) + Math.pow((env.y_tractor - y_weed), 2))
    var d_closest = Math.sqrt(Math.pow((env.x_closest - x_weed),
     2) + Math.pow((env.y_closest - y_weed), 2))
    if(d_tractor < d_closest){
        env.closest_tractor = env.tractor_name;
    }
}
next();
```

Listing 2: *Update Closest* script task

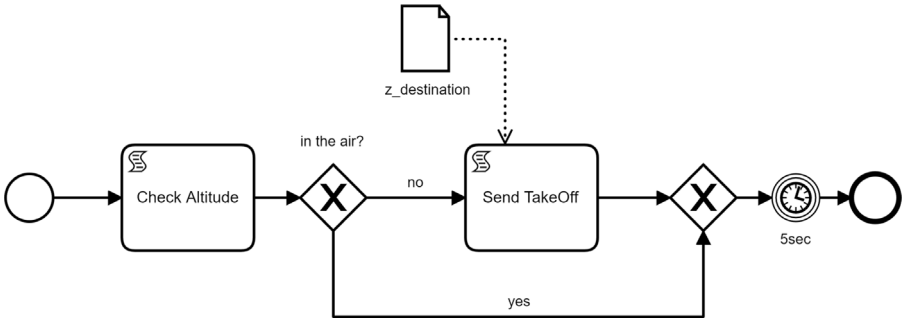


Fig. A.16. *Take off* call activity.

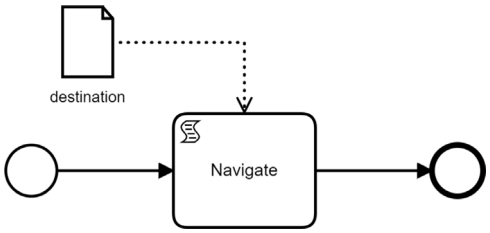


Fig. A.17. *Go To* call activity.

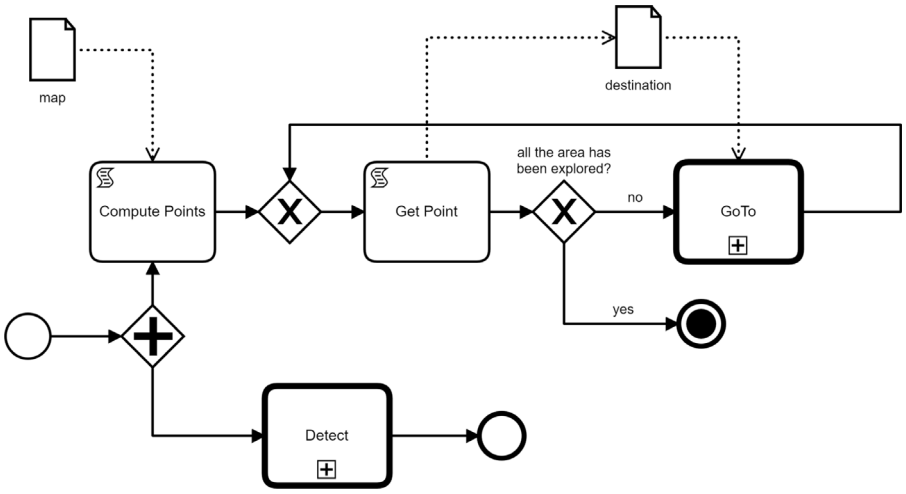


Fig. A.18. *Explore* call activity.

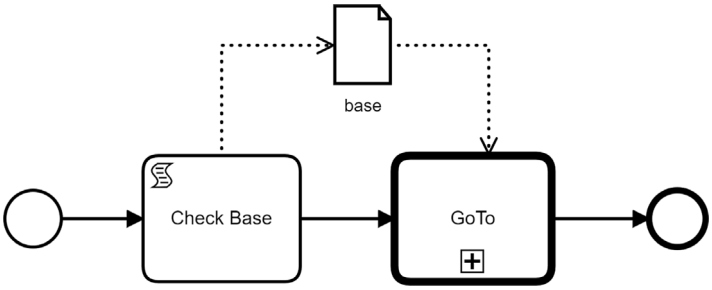


Fig. A.19. *Return to Base* call activity.

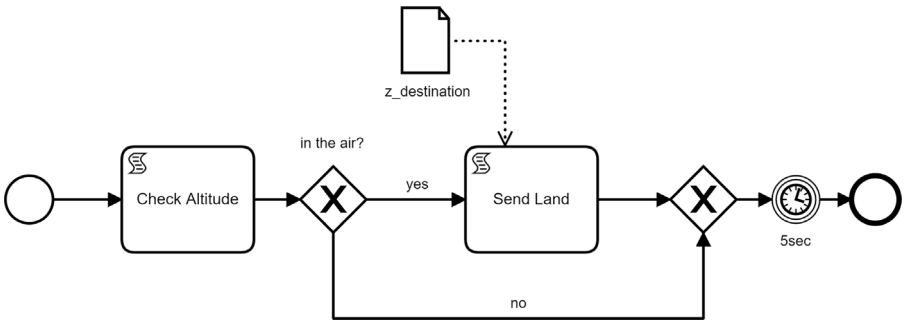


Fig. A.20. *Land* call activity.

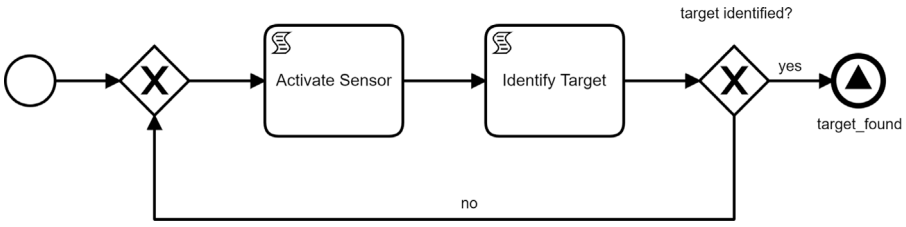


Fig. A.21. *Detect* call activity.

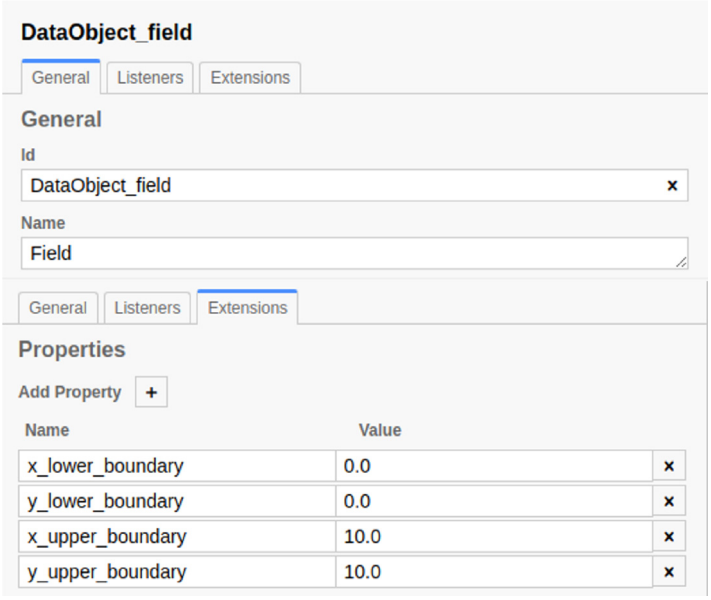


Fig. B.22. Field configuration.

```
const node = this.environment.variables.ros_node;
const subscriber = node.createSubscription("nav_msgs/msg/Odometry
      ", "odom", (msg) => {
    this.environment.variables.my_x = msg.pose.pose.position.x;
    this.environment.variables.my_y = msg.pose.pose.position.y;
    this.environment.variables.my_z = msg.pose.pose.position.z;
    node.destroySubscription(subscriber);
});
next();
```

Listing 3: *Get Position* script task

```
const node = this.environment.variables.ros_node;
const publisher = node.createPublisher("geometry_msgs/msg/Wrench"
      , "blade_force");
publisher.publish({
force: {x: 0.0, y: 0.0, z: 0.0},
torque: {x: 0.0, y: 0.0, z: 1.0}
});
next();
```

Listing 4: *Cut Grass* script task

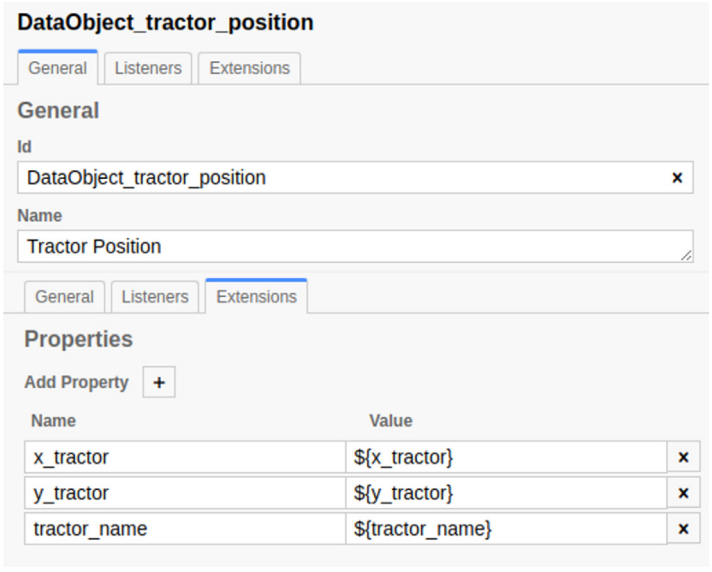


Fig. B.23. Tractor Position configuration.

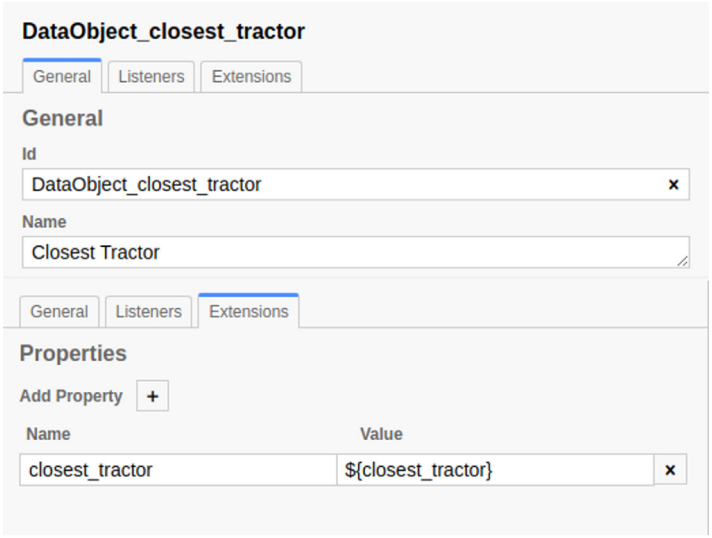


Fig. B.24. Closest Tractor configuration.

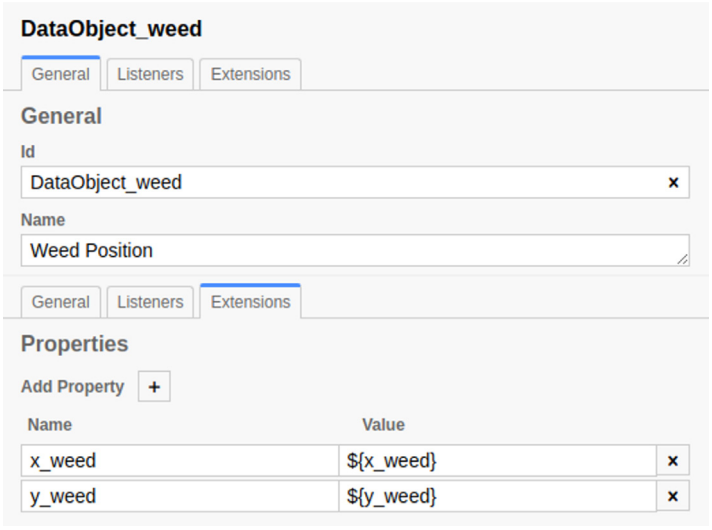


Fig. B.25. Weed Position configuration.

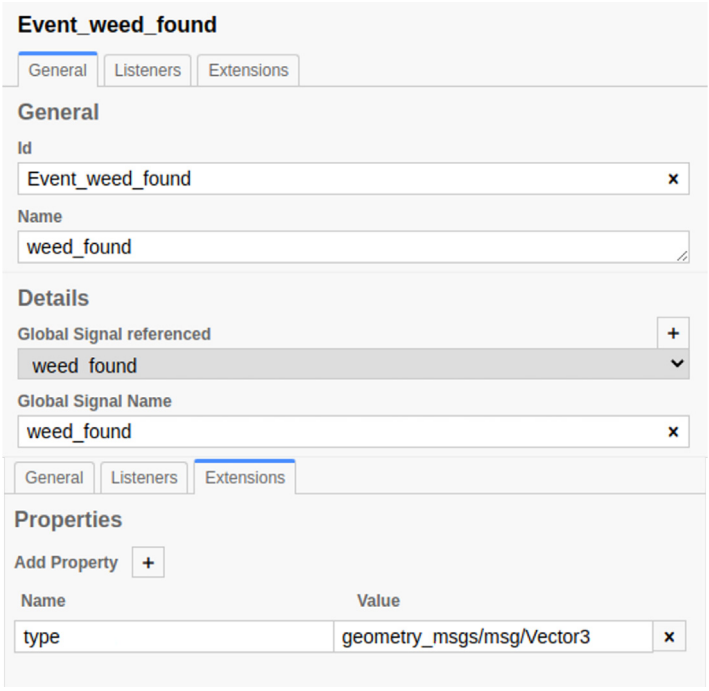
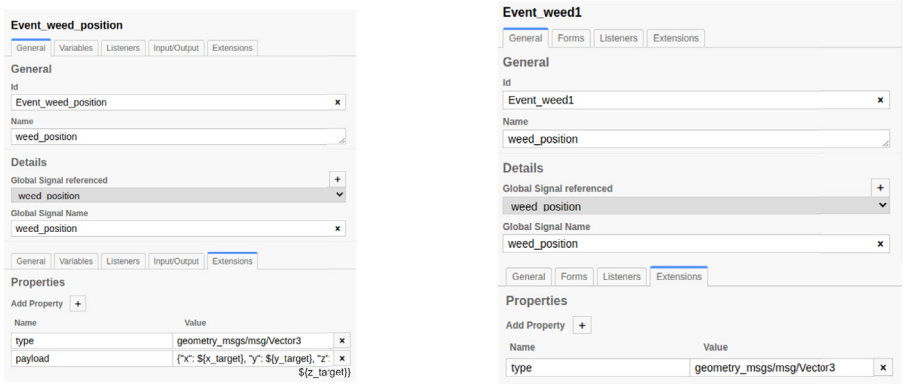
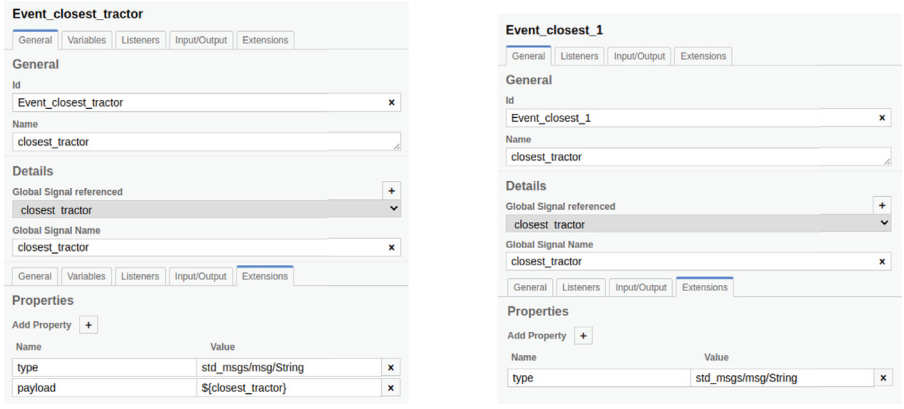


Fig. B.26. *Weed_found* catching configuration.



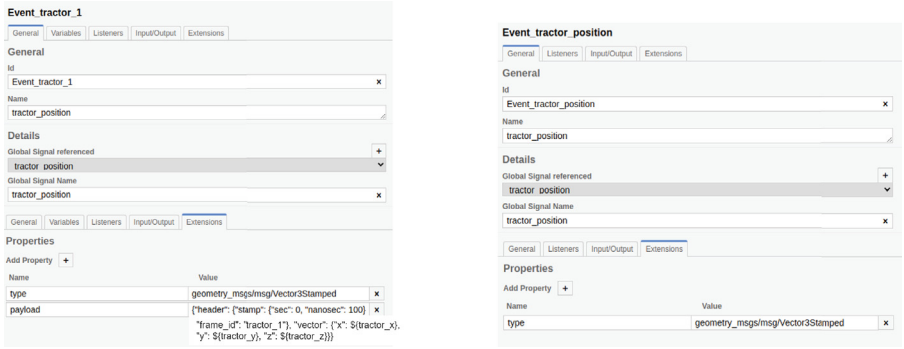
(a) Throwing configuration (b) Catching configuration

Fig. B.27. *Weed_position* signals configuration.



(a) Throwing configuration (b) Catching configuration

Fig. B.28. *Closest_tractor* signals configuration.



(a) Throwing configuration (b) Catching configuration

Fig. B.29. *Tractor_position* signals configuration.

B.2. Data objects configuration

See Figs. B.22–B.25.

B.3. Signals configuration

See Figs. B.26–B.29.

References

[1] C. de Silva, Some issues and applications of multi-robot cooperation, in: Computer Supported Cooperative Work in Design, IEEE, 2016, p. 2.

[2] T. Arai, E. Pagello, L. Parker, Guest editorial advances in multirobot systems, IEEE Trans. Robot. Autom. 18 (5) (2002) 655–661.

[3] Y. Maruyama, S. Kato, T. Azumi, Exploring the performance of ROS2, in: Proceedings of the International Conference on Embedded Software, ACM, 2016, pp. 1–10.

[4] OMG, Data distribution service (DDS) v. 1.4, 2015.

[5] C. Crick, G. Jay, S. Osentoski, B. Pitzer, O. Jenkins, Rosbridge: ROS for non-ROS users, in: Robotics Research : The 15th International Symposium ISRR, in: STAR, vol. 100, Springer, 2017, pp. 493–504.

[6] A. Nordmann, N. Hochgeschwender, S. Wrede, A survey on domain-specific languages in robotics, in: SIMPAR, in: LNCS, vol. 8810, Springer, 2014, pp. 195–206.

[7] E. de Araújo Silva, E. Valentin, J. Carvalho, R. da Silva Barreto, A survey of model driven engineering in robotics, Comput. Lang. 62 (2021) 101021.

[8] K. Bourr, F. Corradini, S. Pettinari, B. Re, L. Rossi, F. Tiezzi, Disciplined use of BPMN for mission modeling of Multi-Robot Systems, in: Proceedings of the Forum at Practice of Enterprise Modeling, Vol. 3045, CEUR-WS.org, 2021, pp. 1–10.

[9] OMG, Business process model and notation (BPMN) v. 2.0, 2011.

[10] F. Corradini, C. Muzi, B. Re, L. Rossi, F. Tiezzi, Formalising and animating multiple instances in BPMN collaborations, Inf. Syst. 103 (2022) 101459.

[11] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, M. Tivoli, FLYAQ: Enabling non-expert users to specify and generate missions of autonomous multicopters, in: ASE, 2015, pp. 801–806.

[12] F. Ciccozzi, D. Di Ruscio, I. Malavolta, P. Pelliccione, Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems, IEEE Access 4 (2016) 6451–6466.

[13] M. Kocbek, G. Jost, M. Hericko, G. Polancic, Business process model and notation: The current state of affairs, Comput. Sci. Inf. Syst. 12 (2) (2015) 509–539.

[14] D. Kozma, P. Varga, F. Larrinaga, Data-driven workflow management by utilising BPMN and CPN in IIoT systems with the arrowhead framework, in: International Conference on Emerging Technologies and Factory Automation, IEEE, 2019, pp. 385–392.

[15] R. Woitsch, W. Utz, A. Sumereder, B. Dieber, B. Breiling, L. Crompton, M. Funk, K. Bruckmüller, S. Schumann, Collaborative model-based process assessment for trustworthy AI in robotic platforms, in: Society 5.0, Springer International Publishing, 2021, pp. 163–174.

[16] R. Rey, M. Corzetto, J.A. Cobano, L. Merino, F. Caballero, Human-robot co-working system for warehouse automation, in: International Conference on Emerging Technologies and Factory Automation, IEEE, 2019, pp. 578–585.

[17] J.-P. de la Croix, G. Lim, Event-driven modeling and execution of robotic activities and contingencies in the europa lander mission concept using BPMN, in: I-SAIRAS, ESA, 2020.

[18] O. Kyohei, T. Scott, T. Rohan, T. Vaquero, E. Jeffrey, W. William, M. Gregory, H. Tristan, W. Michael, A.-M. Ali-Akbar, Supervised autonomy for communication-degraded subterranean exploration by a robot team, in: Aerospace Conference, IEEE, 2020, pp. 1–9.

[19] J. López, P. Sánchez-Vilariño, R. Sanz, E. Paz, Implementing autonomous driving behaviors using a message driven Petri-net framework, Sensors 20 (2020) 449.

[20] M. Figat, C. Zielinski, Robotic system specification methodology based on hierarchical Petri nets, IEEE Access 8 (2020) 71617–71627.

[21] M. Figat, C. Zieliński, Parameterised robotic system meta-model expressed by hierarchical Petri nets, Robot. Auton. Syst. 150 (2022) 103987.

[22] J. Harbin, S. Gerasimou, N. Matragkas, A. Zolotas, R. Calinescu, Model-driven simulation-based analysis for multi-robot systems, in: Model Driven Engineering Languages and Systems, IEEE, 2021, pp. 331–341.

[23] T. Morita, T. Yamaguchi, Generating ROS codes from user-level workflow in PRINTEPS, in: Domain-Specific Conceptual Modeling: Concepts, Methods and ADOxx Tools, Springer, 2022, pp. 435–455.

[24] I. Compagnucci, F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, Modelling notations for IoT-aware business processes: A systematic literature review, in: BP-Meet-IoT, in: LNCS, 397, Springer, 2020, pp. 108–121.

[25] F. Corradini, C. Muzi, B. Re, F. Tiezzi, L. Rossi, MIDA: multiple instances and data animator, in: BPM Demos, Vol. 2196, CEUR-WS.org, 2018, pp. 86–90.

[26] F. Corradini, A. Morichetta, A. Polini, B. Re, L. Rossi, F. Tiezzi, Correctness checking for BPMN collaborations with sub-processes, J. Syst. Softw. 166 (2020) 110594.

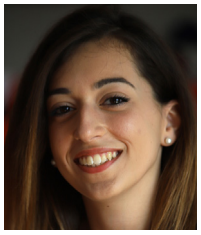
[27] M. Iftikhar, J. Lundberg, D. Weyns, A model interpreter for timed automata, in: International Symposium on Leveraging Applications of Formal Methods, in: LNCS, vol. 9952, 2016, pp. 243–258.

[28] N. Ritschel, V. Kovalenko, R. Holmes, R. Garcia, D.C. Shepherd, Comparing block-based programming models for two-armed robots, IEEE Trans. Softw. Eng. (2020) 1.

[29] M.E.A. Chergui, S.M. Benslimane, Towards a BPMN security extension for the visualization of cyber security requirements, Int. J. Technol. Diffus. (IJTD) 11 (2) (2020) 1–17.

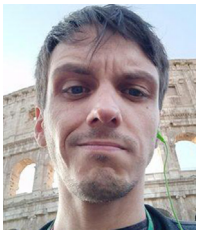
[30] F. Corradini, B. Re, L. Rossi, F. Tiezzi, A technique for collaboration discovery, in: International Conference on Business Process Modeling, Development and Support, in: LNBIP, vol. 450, Springer, 2022, pp. 63–78.

Flavio Corradini is Full Professor of Computer Science at the University of Camerino. He received a Laurea Degree in Computer Science from the University of Pisa and a Ph.D. in Computer Engineering from the University of Rome "La Sapienza". He has been Director of the Mathematics and Computer Science Department (2006–2009), President of the center for digital services and information systems of the University of Camerino (2004–2010), Coordinator of the Computer Science Studies of the University of Camerino (2004–2006), and rector of the University of Camerino (2010–2016). His main research activities are in the area of formal specification, verification of concurrent, distributed and real-time systems.



Sara Pettinari is a Ph.D. Candidate in Computer Science and Mathematics at the University of Camerino. She has received a Bachelor and Master Degree at the University of Camerino. Her research interests are on modeling and development of IoT and Robotic systems, with a focus on the analysis of their behaviors and interactions.

Barbara Re is Associate Professor of Computer Science at the University of Camerino. She received her Ph.D. in Information Science and Complex System from University of Camerino. Her research interests refer to the area of Business Process Management from modeling to analysis. Particular attention is paid to push the use of formal methods as methodological and automatic tools for the development of high-quality process-aware information system. She was involved in multidisciplinary research projects in collaboration with national and international research institutes and companies.



Lorenzo Rossi is a Post-doc at the University of Camerino (Italy), where he earned a Ph.D. in Computer Science in 2019. He also won different scholarships, at the same institution, for conducting research on Business Process Animation. His main research interests are in the fields of Business Process Management and Software Engineering. He aims at providing software solutions, with a solid base on theoretical computer science, for the management of business processes.

Francesco Tiezzi is Associate Professor of Computer Science at the University of Florence. He received the Laurea degree from University of Florence and the Ph.D. degree from the same university. His research activities focus on modeling and programming languages, foundational study of distributed systems, and application of formal methods for developing and analyzing them. Particular attention is paid to the definition of formal bases for technologies supporting service-oriented, cloud and autonomic computing.