International Journal on Software Tools for Technology Transfer (2023) 25:747–764
https://doi.org/10.1007/s10009-023-00727-w

FOUNDATIONS FOR MASTERING CHANGE

Special Section: Rigorous Engineering of Collective Adaptive Systems

Coordinating and programming multiple ROS-based robots with X-KLAIM

Lorenzo Bettini[1] · Khalid Bourr[2] · Rosario Pugliese[1] · Francesco Tiezzi[1]

Accepted: 10 October 2023 / Published online: 2 November 2023

Abstract

Software development for robotics applications is still a major challenge that becomes even more complex when considering multi-robot systems (MRSs). Such distributed software has to perform multiple cooperating tasks in a well-coordinated manner to avoid unsatisfactory emerging behavior. This paper provides an approach for programming MRSs at a high abstraction level using the programming language X-KLAIM. The computation and communication model of X-KLAIM, based on multiple distributed tuple spaces, permits coordinating with the same abstractions and mechanisms both intra- and inter-robot interactions of an MRS. This allows developers to focus on MRS behavior, achieving readable, reusable, and maintainable code. The proposed approach can be used in practice by integrating X-KLAIM and the popular robotics framework ROS. We demonstrate the feasibility and effectiveness of our approach by (i) showing how it scales when implementing two warehouse scenarios allowing us to reuse most of the code when passing from the simpler to the more enriched scenario and (ii) presenting the results of a few experiments showing that our code introduces a slightly greater but acceptable latency and consumes less memory than the traditional ROS implementation based on Python code.

Keywords Multi-robot systems · Multiple tuple spaces · X-Klaim · ROS

1 Introduction

Autonomous robots are software-intensive systems and are increasingly used in many different fields. Their software components interact in real-time with a highly dynamic and uncertain environment through sensors and actuators. To complete tasks that are beyond the capabilities of an individual autonomous robot, multiple robots are teamed together to form a *multi-robot system* (MRS). An MRS can take advantage of distributed sensing and action and has greater reliability. However, an MRS requires robots to cooperate and coordinate to achieve common goals.

The development of the software controlling a single autonomous robot is still a challenge [1–3]. This becomes even more arduous in the case of MRSs [4, 5], as it requires dealing with multiple cooperating tasks to drive the robots to work as a well-coordinated team. To meet this challenge, various software libraries, tools, and middlewares have been proposed to assist in and simplify the rapid prototyping of robotics applications. Among them, nowadays, a prominent solution is the Robot Operating System (ROS) [6], a popular framework largely used in both industry and academia for writing robot software. On the one hand, ROS provides a layer to interact with many sensors and actuators for many robots while abstracting from the underlying hardware. On the other hand, programming with ROS still requires dealing with low-level implementation details; hence, robotics software development remains a complex and demanding activity for practitioners from the robotic domain. To overcome this issue, many researchers have proposed using higher-level abstractions to drive the software development process and then resorting to tools for the automated generation of executable code and system configuration files. Many proposals in the literature are surveyed in [3, 7–9].

✉ F. Tiezzi
francesco.tiezzi@unifi.it

L. Bettini
lorenzo.bettini@unifi.it

K. Bourr
khalid.bourr@unicam.it

R. Pugliese
rosario.pugliese@unifi.it

[1] Dipartimento di Statistica, Informatica, Applicazioni, Università degli Studi di Firenze, Viale Morgagni, 65, Firenze, 50134, Italy

[2] School of Science and Technology, Università di Camerino, Via Madonna delle Carceri, 7, Camerino, 62032, Italy

Along this line of research, we introduced in [10] an approach for programming a single-robot system. Specifically, we propose using the language X-KLAIM [11] to program the components of a robot's software. This choice is motivated by the fact that X-KLAIM provides mechanisms based on distributed tuple spaces for coordinating the interactions between these software components at a high level of abstraction. The integration of X-KLAIM with ROS permits the application of the approach in practice.

In [12], we took a first step forward in this direction by extending the approach in [10] to program MRSs. In fact, the X-KLAIM computation and communication model is particularly suitable for dealing with both (i) the distributed nature of the architecture of each robot belonging to an MRS, where the software components dealing with actuators and sensors execute concurrently, and (ii) the inherent distribution of the MRS, which is formed by multiple interacting robots. Notably, the same tuple-based mechanisms are used for both intra- and inter-robot communication. This simplifies the design and implementation of MRS software in terms of an X-KLAIM application distributed across multiple threads of execution and hardware platforms, resulting in better readable, maintainable, and reusable code.

In this paper, we build on our previous work [12] proposing an improved programming methodology that structures the implementation of the behavior of single robots into different layers with clearly separated responsibilities. This allowed us to produce clearer, more reusable, and more maintainable code. We illustrate our approach by implementing two warehouse scenarios involving an MRS that manages the movement of items through an arm robot and one or more delivery robots. The two scenarios show how the new programming methodology enables incremental development of the MRS by reusing most of the code when scaling from the first, simpler scenario to the more enriched one.

Our framework can be considered a proof-of-concept implementation for experimenting with the applicability of the tuple space-based paradigm to MRS software development. To show the execution of the generated code, we use a simulator of robot behaviors.

We also present the results of a few experiments aimed at determining the impact of our software framework on the performance of MRSs. The experiments compare the time and memory performance of our implementation of the MRS of the warehouse scenarios based on Java code and ROS Bridge against the traditional ROS implementation based on Python code. The experiments show that our Java code introduces a slightly greater but acceptable latency and consumes less memory than the Python code.

Contribution This paper extends our previous work [10, 12] in several ways.

- We provide a more detailed description of the languages and technologies on which our approach is based.
- We adopt an improved programming methodology prescribing to structure the implementation of the behavior of single robots into different layers with clearly separated responsibilities, thus enabling code reuse.
- We provide a reimplementation of the warehouse example in [12] that takes advantage of the incremental development fostered by our programming methodology.
- We present the results of a few experiments to measure the impact of our framework on the performance of MRSs.

Structure of the paper The rest of the paper is organized as follows. In Sect. 2, we recall some background notions on the languages and technologies we use in our approach, while in Sect. 3, we describe our approach and software framework. In Sect. 4, we show and briefly comment on our approach at work on implementing two warehouse scenarios involving an MRS that manages the movement of items. In Sect. 5, we illustrate the experiments we carried out to determine the impact of our approach on MRS performance. In Sect. 6, we present a systematic analysis of more strictly related work, while in Sect. 7, we conclude and touch upon directions for future work.

2 Background notions

In this section, we recall a few background notions on the languages and technologies we use in our approach. We refer the interested reader to the cited sources for a complete account.

2.1 Klaim

KLAIM (Kernel Language for Agents Interaction and Mobility, [13]) is a formal language specially devised to design distributed applications consisting of possibly mobile software components deployed over the nodes of a network infrastructure. KLAIM is based on process calculi [14]. It generalizes the notion of *generative communication*, introduced by the coordination language Linda [15], to multiple distributed tuple spaces. A *tuple space* is a shared data repository consisting of a multiset of tuples. *Tuples* are *anonymous* sequences of data items that are associatively retrieved from tuple spaces using a *pattern-matching* mechanism. Communicating processes are decoupled in both space and time as there is no need for producers (i.e., senders) and consumers (i.e., receivers) of a tuple to synchronize. Inter-process communication occurs through the *asynchronous* exchange of tuples via tuple spaces: processes can indeed *insert*, *read*, and *withdraw* tuples into/from tuple spaces. Tuple spaces are identified through *localities*, which are symbolic addresses

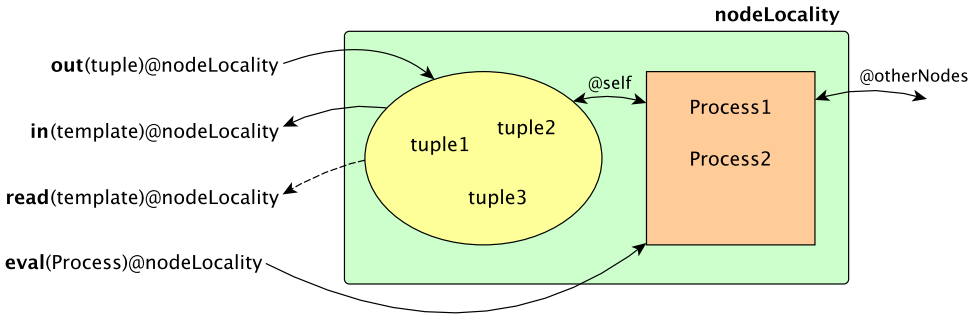

Fig. 1 A KLAIM node

of network nodes where processes and tuples can be allocated. Localities themselves can be exchanged through interprocess communication.

A computational *node* of a KLAIM network is characterized by its locality and a collection of running processes. *Processes* are the active computational units of KLAIM and can be executed concurrently, either at the same locality or at different localities. Processes can execute basic actions acting on network nodes, process variables, and process calls, either sequentially or in parallel. KLAIM supports *higher-order communication* since processes can exchange code and possibly execute it. Recursive behaviors are modeled via calls to process definitions.

Figure 1 depicts a generic KLAIM node and the basic *actions* which processes are made of. In these actions, processes can use the distinguished locality self to refer to their current hosting node.

Action **out**(tuple)@nodeLocality adds the tuple resulting from the evaluation of the argument tuple to the tuple space of the target node identified by the (possibly remote) locality nodeLocality. A tuple is a sequence of *actual* fields, i.e., expressions, localities, or processes. In general, any of these fields can contain variables. The evaluation of a tuple consists of evaluating the expressions it contains. Hence an evaluated tuple cannot contain variables.

Action **in**(template)@nodeLocality (resp., **read**(template)@nodeLocality) withdraws (resp., reads) tuples from the tuple space hosted at the (possibly remote) locality nodeLocality. If matching tuples are found, one is nondeterministically chosen. Otherwise, the process is blocked until a matching tuple is found. These retrieval actions exploit templates as patterns to select tuples in tuple spaces. *Templates* are sequences of actual and *formal* fields, where the latter are used to bind variables to values, localities, or processes. Templates must be evaluated before they can be used for retrieving tuples. Their evaluation is like that of tuples, where the evaluation leaves formal fields unchanged. Intuitively, an evaluated template matches against an evaluated tuple if both have the same number of fields and corresponding fields match; two values/localities match only if they are identical, while formal fields match any value of the same type. Upon a successful matching, the template variables are replaced with the values of the corresponding actual fields of the accessed tuple.

Action **eval**(Process)@nodeLocality sends Process for execution to the (possibly remote) node identified by nodeLocality.

2.2 KLAVA and X-KLAIM

The implementation of KLAIM consists of two main components:

- the Java package KLAVA;
- the programming language X-KLAIM.

KLAVA (KLAIM in Java, originally introduced in [16]) provides the implementation of the KLAIM concepts (Sect. 2.1) in terms of Java classes and methods, relying on the IMC framework [17] for the communication infrastructure. Any Java object can be stored into and retrieved from a KLAVA tuple, and the implemented pattern-matching mechanism keeps Java subtyping into consideration. KLAVA strives to make Java programmers' life easier, but programmers still have to obey the rules of Java, particularly its verbosity. For this reason, we also developed X-KLAIM, a domain-specific language (DSL) closer to KLAIM also providing typical high-level programming constructs. X-KLAIM (eXtended KLAIM) was initially introduced in [18] and reimplemented from scratch in [11]. The X-KLAIM compiler translates X-KLAIM programs into Java code that uses the Java package KLAVA. The produced Java code can then be compiled and executed using the standard Java toolchain.

The new implementation of X-KLAIM [11] is based on XTEXT [19], an Eclipse framework for developing programming languages and DSLs. XTEXT also provides complete IDE support based on Eclipse: editor with syntax highlighting, code completion, error reporting, and incremental building, to mention a few. Furthermore, we used another mechanism provided by XTEXT, that is, XBASE [20], an extensible and reusable expression language. By using XBASE

in X-KLAIM, besides a rich Java-like syntax, we inherit its interoperability with Java and its type system. Thus, an X-KLAIM program can smoothly access any Java type and Java library available in the project's classpath. The interoperability with Java allowed us to integrate X-KLAIM seamlessly with the Java–ROS connector (see Sect. 3).

In the rest of this section, we briefly describe the main features of X-KLAIM relevant to this paper.

An X-KLAIM program (a file with extension `.xklaim`) can contain definitions of nets, nodes, and processes. All these components can also be defined in separate files and referred to through a Java-like *import* mechanism. As in a standard Java program, imports are also used to import existing Java types in an X-KLAIM program, relying on the integration with Java mentioned above.

An X-KLAIM network definition consists of *net* and *node* definitions as shown in the following example:

```
net ANet {
    node Node1 { ... initialization code ... }
    node Node2 { ... initialization code ... }
    ...
}
```

In particular, the name of a node also represents its locality within the network. Each node can specify some initialization code for creating and running a few processes, as shown in the examples of Sect. 4. This is the simplest way of specifying a *flat* network. X-KLAIM also implements the hierarchical version of the KLAIM model as presented in [21], but we will not use it in this paper.

A *process* definition consists of a name, a list of parameters (using the Java syntax for declaring parameters), and a body:

```
proc AProcess(... parameters ...) { ... body ... }
```

The body consists of XBASE expressions, whose syntax has been extended with the KLAIM operations that we described in Sect. 2.1. Typical programming structures such as `if` and `while` and OOP Java-like mechanisms such as object creation and method invocation are already part of XBASE.

The syntax of XBASE is similar to that of Java, and it should be easily understood by Java programmers, but it lacks much "syntactic noise" from Java. For example, terminating semicolons and other syntax elements like parentheses when invoking a method without arguments are optional. Moreover, XBASE comes with a powerful type inference mechanism compliant with the Java type system: the programmer can thus avoid specifying types in declarations when they can be inferred from the context. Variable declarations start with `val` or `var` for final and non-final variables, respectively. The type of a variable can be omitted if it can be inferred from the initialization expression.

```
in("item", var String itemId,
    var Double x, var Double y)@self
System.err.println("Coordinates: " + x + ", " + y)
out(itemId, x, y)@otherLoc
eval(new AProcess(x,y))@self
val strings = #["first", "second", "third"]
strings.stream().map([ s | s.length()])
    .forEach([ l | System.err.println(l) ])
```

Fig. 2 An example of X-KLAIM code

In Fig. 2, we show a simple code snippet of an X-KLAIM process body. The code should be easily readable by a Java programmer. We mention a few additional X-KLAIM syntax features to make the code more understandable. Such types as `String` and `Double` are Java types since, as mentioned above, X-KLAIM programs can refer directly to Java types. Similarly, `System.err.println` is the standard Java static method to print something on the screen. In most code snippets, we omit the Java-like import statements. Here we also see the typical KLAIM operations `in` and `out`, acting on possibly distributed tuple spaces. Formal fields in a tuple are specified as variable declarations since formal fields implicitly declare variables that are available in the code occurring after `in` and `read` operations (just like in KLAIM). The X-KLAIM operation `eval` allows a process to start a new process concurrently at the specified locality. X-KLAIM provides syntactic sugar for collection literals: `#[ ... ]`. In the code snippet, `strings` is inferred to be of type `List<String>`. In X-KLAIM, *lambda expressions* have the shape `[ param1, param2, ... | body ]`, where the types of parameters can be omitted if they can be inferred from the context. An X-KLAIM lambda expression can be used in any Java context where a lambda expression can be used, according to the Java type system: the type of the lambda expression must match the target Java functional interface. The code snippet shows standard Java stream operations using X-KLAIM lambda expressions.

The X-KLAIM compiler is completely integrated into Eclipse: typical IDE mechanisms like content assist and code navigation are available in the X-KLAIM editor. The same holds for the automatic building mechanisms of Eclipse: saving an X-KLAIM file automatically triggers the Java code generation, which in turn triggers the generation of Java byte code. From a single X-KLAIM program, our compiler generates several Java classes (e.g., one for a net, one for each node, and one for each process) that extend and use KLAVA classes. The relation between X-KLAIM elements and the generated Java classes is handled transparently. For example, removing a process from an X-KLAIM program will automatically remove the previously generated corresponding Java class.

Finally, the X-KLAIM Eclipse support also includes the ability to directly run or debug an X-KLAIM file with dedicated context menus: there is no need to run the generated

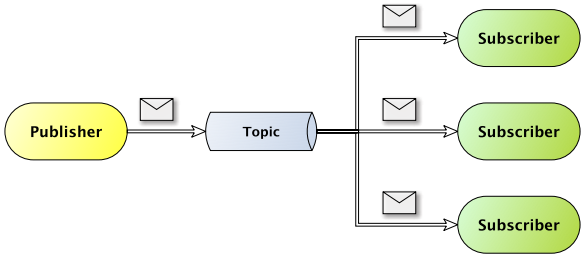

Fig. 3 ROS publish/subscribe mechanism

Java code manually. Debugging an X-KLAIM program directly is crucial when programming distributed applications accessing remote tuple spaces. We can set a breakpoint in the X-KLAIM program, possibly based on a condition. During the execution of the corresponding generated Java code, the execution is suspended on the X-KLAIM program: we can inspect the current values of variables, either in the "Variables" Eclipse view or by hovering over a variable in the program. The debugging mechanisms of X-KLAIM are as powerful as Eclipse's standard Java debugging mechanism. For example, during an X-KLAIM debugging session, we can evaluate expressions on the fly.

2.3 ROS

ROS[1] is one of the most sophisticated and popular frameworks for writing robot software. It provides tools and libraries for simplifying the development of complex controllers while abstracting from the underlying hardware. ROS works with more than 100 types of robots, ranging from autonomous cars to drones and humanoid robots, and integrates many sensors.

The core element of the ROS framework is the message-passing middleware, which enables hardware abstraction for a wide variety of robotic platforms. The processes of a robotics application can exchange data, being agnostic with respect to the source of the data. The communicated data can be sensor readings or actuator commands, formatted in a standardized way, produced by or directed to the robot's devices.

Although ROS supports different communication mechanisms, in this paper, we only use the most common one: the anonymous and asynchronous publish/subscribe mechanism. To send a message, a process has to publish it in a *topic*, which is a named and typed bus. A process interested in such a message must subscribe to the topic. The subscriber will be notified whenever a new message is published on the topic. This decouples the production of data from its consumption.

Multiple publishers and subscribers for the same topic are allowed. The diagram in Fig. 3 illustrates this concept.

[1] https://www.ros.org/.

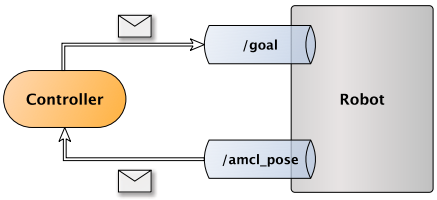

Fig. 4 Interactions within a mobile ROS robot

In contrast, the one in Fig. 4 shows how a robot controller interacts with the devices of a mobile robot in a black-box, hardware-independent fashion. In the latter diagram, the controller acts as both publisher and subscriber. As a publisher, it sends a message directed to the navigation node responsible for actuating the wheels. At the same time, as a subscriber, it receives back messages containing the robot's actual position. The topic `/goal` stands for the *goal* position that the mobile robot should attempt to reach. The topic `/amcl_pose` stands for the estimated *pose* of the robot, in a known map, calculated from the robot sensor data with the "adaptive Monte Carlo localization" (AMCL) approach.

3 The X-KLAIM approach to multi-robot programming

In this section, we describe our approach and software framework for programming MRS applications based on integrating ROS and X-KLAIM.

A single autonomous robot has a distributed architecture consisting of cooperating components, particularly sensors and actuators. Such cooperation is enabled and controlled by the ROS framework.

When passing from a single-robot system to an MRS, the distributed and heterogeneous nature of the overall system becomes even more evident. The software architecture for controlling an MRS reflects such a distribution: each robot is equipped with ROS, on top of which the controller software runs. This allows the robot to act independently and, when needed, to coordinate with the other robots of the system to work together coherently.

In X-KLAIM, the distributed architecture of the MRS's software is naturally rendered as a network where the different parts are deployed. As shown in Fig. 5, we associate an X-KLAIM node to each robot of the MRS. In its turn, the internal distribution of the software controller of each robot is managed by concurrent processes that synchronize their activities using tuples stored in the robot's tuple space. Inter-robot interactions rely on the same communication mechanism by specifying remote tuple spaces as targets of communication actions.

In this paper, we prescribe that processes related to the behavior of a single robot can be structured in different logical

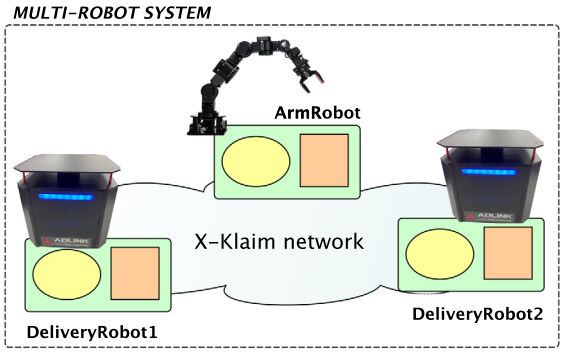

Fig. 5 Software architecture of an MRS in X-KLAIM

layers to clearly separate their responsibilities. The top layer defines the lifecycle of the robot's behavior, which is typically expressed as a process that cyclically performs a main macro activity. In a second layer, we specify the logic of the macro activity by coordinating a specific robot's activities, expressed as processes interacting with the robot's physical devices. These latter processes are the building blocks of the programming approach and form the bottom layer. All layers' processes can be parameterized to be reusable in the same or different robotics applications.

In practice, to program the behaviors of the robots forming an MRS, we enabled X-KLAIM programs to interact with robots' physical components by integrating the X-KLAIM language with the ROS middleware. The communication infrastructure of the integrated framework, graphically depicted in Fig. 6, is based on ROS Bridge. This server is included in the ROS framework and provides a JSON API to ROS functionalities for external programs. This way, the ROS framework installed in a robot receives and executes commands on the physical components of the robot and gives feedback and sensor data.

The use of JSON enables the interoperability of ROS with most programming languages, including Java. As an example, we report in Fig. 7 a message `pose` in the JSON format published on the ROS topic `/goal`, providing information for navigating a delivery robot to a given goal position. In our example, the goal is the position $(-0.21, 0.31)$.

X-KLAIM programs can indirectly interact with the ROS Bridge server, publishing and subscribing over ROS topics via objects provided by the Java library *java_rosbridge*.[2] In its own turn, *java_rosbridge* communicates with the ROS Bridge server via the WebSocket protocol through the Jetty web server.[3]

ROS permits checking the execution of the code generated from an X-KLAIM program by means of the Gazebo[4] simulator. Gazebo [22] is an open-source simulator of robot behaviors in complex environments that is based on a robust physics engine and provides a high-quality 3D visualization of simulations. Gazebo is fully integrated with ROS; in fact, ROS can interact with the simulator via the publish/-subscribe communication mechanism of the framework. The use of the simulator is not mandatory when ROS is deployed in real robots. However, even in such a case, the MRS software design activity may benefit from using a simulator to save time and reduce development costs.

Since the X-KLAIM compiler generates plain Java code, which depends only on KLAVA and a few small libraries, an X-KLAIM application can be deployed by using standard Java tools and mechanisms. It is enough to create a JAR with the generated Java code and its dependencies (KLAVA and *java_rosbridge*), that is, a so-called "fat-jar" or "uber-jar". Such a JAR file can be deployed to every physical robot where a Java virtual machine is already installed. In that respect, X-KLAIM provides standard Maven artifacts and a plugin to generate Java code outside Eclipse, e.g., in a continuous integration server. Moreover, the dependencies of an X-KLAIM application, including *java_rosbridge*, are only a few megabytes, which makes X-KLAIM applications suitable also for embedded devices like robots.

4 The X-KLAIM approach at work on MRS scenarios

To illustrate the proposed approach, in this section, we show and briefly comment on a few interesting parts of implementing two warehouse scenarios[5] involving an MRS that manages the movement of items.

In Sect. 4.1, we present a simple warehouse scenario with an MRS composed of an arm robot and a delivery robot working together in an environment free of obstacles. Then, in Sect. 4.2, we show an enriched version of the scenario with multiple delivery robots and a more realistic warehouse environment, focusing on how the code of Sect. 4.1 can be reused and extended in only a few parts.

4.1 Simple warehouse scenario

This first simple scenario comprises an arm robot and a delivery robot. The arm robot, positioned in the center of the warehouse, picks up one item from the floor, calls the delivery robot, and releases the item on top of the delivery robot. The delivery robot delivers the item to the appropriate delivery area and becomes available for a new delivery.

[2] https://github.com/h2r/java_rosbridge.

[3] Jetty 9: https://www.eclipse.org/jetty/.

[4] https://gazebosim.org/.

[5] The complete source code of the scenarios' implementation can be found at https://github.com/LorenzoBettini/xklaim-ros-warehouse-scenarios.

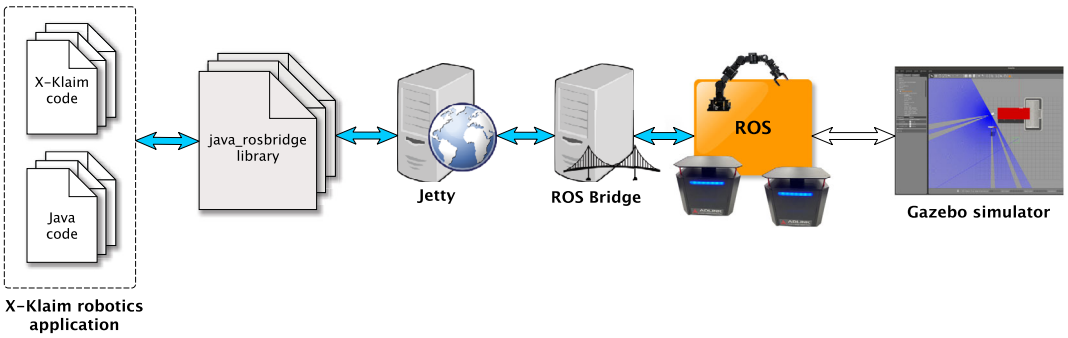

Fig. 6 The integrated framework

```
{"topic": "/robot1/move_base_simple/goal",
 "msg": {"header": { ... },
         "pose": {"position": {"x": −0.21, "y": 0.31,
                               "z": 0.0 },
                  "orientation": { ... } } } }
```

Fig. 7 Example of a JSON message for the `/goal` topic

```
net MRS_one_delivery physical "localhost:9999" {
    node Arm {
        eval(new ArmBehavior())@self
    }

    node DeliveryRobot {
        val robotId = "robot"
        val sector = "sector"
        eval(new DeliveryRobotBehavior
      (robotId, sector, Arm))@self
    }

    node Environment {...}
}
```

Fig. 8 The X-KLAIM net of the simple warehouse scenario

In Fig. 8, we show a part of the network for our scenario implementation. Each robot is rendered as an X-KLAIM node, whose name represents its locality (see Sect. 2).

Each node creates its process locally, representing the node/robot behavior, and executes it concurrently using the X-KLAIM operation `eval`. The delivery robot process behavior is parametric concerning the identifier and the sector. These parameters facilitate the reusability of the process, as shown in Sect. 4.2. The node `Environment` provides an interface with the simulated environment. In this scenario, it notifies the arm robot about the presence of items to be picked up and consumes them when delivered to the destination. While these actions are simulated here, in a real-world implementation of the scenario they might be performed by physical devices or human actors.

In this paper, the processes representing robot behavior have a common shape. As an example, we show the process of the arm behavior in Fig. 9. The idea is that the behavior process defines the lifecycle while the actual implementation logic is delegated to another process (`PickAndReleaseOneItem`, in this case). The process responsible for implementing the logic is meant to be usable with different behaviors. In this example, all the behaviors are recursive. In fact, in Fig. 9, after the execution of the implementation logic, the `ArmBehavior` evaluates another instance of the behavior. Hence, before starting a new instance of the behavior, it has to coordinate with `PickAndReleaseOneItem`. The latter is expected to put a tuple in the local tuple space with an agreed string. To avoid possible spelling mistakes when using constants in tuples, we define the constants in a Java class `ArmConstants`. Note that thanks to the integration with Java (see Sect. 2.2), X-KLAIM can use Java constants with standard "import static" mechanisms. Recall that `eval` spawns another concurrent process and is a non-blocking operation. Thus, the currently running behavior process `ArmBehavior` terminates after starting another instance of itself.

```
import static xklaim.arm.ArmConstants.*

proc ArmBehavior() {
    eval(new PickAndReleaseOneItem())@self
    in(IS_IN_THE_INITIAL_POSITION)@self
    eval(new ArmBehavior())@self
}
```

Fig. 9 The process representing the arm robot behavior

In Fig. 10, we show the code of the process `PickAndReleaseOneItem`. This process waits for a tuple with information concerning an item available for delivery (in our implementation this is provided by the node `Environment` of Fig. 8). Then, it defines a few constants representing trajectories. Trajectories are implemented with plain Java objects and contain a few double numbers corresponding to physical points in the scenario, some of which depend on the item coordinates. We do not show them here because they are not relevant to the aim of this section. The actual logic is implemented by relying on a few reusable processes: `MoveArm` and `UseGripper`, whose names and usages should be self-explanatory. Both processes are parameterized with a trajectory structure, which represents the actual movement, and are started, once again, with `eval`. They notify the completion of their task by out-

```
proc PickAndReleaseOneItem() {
    in(ITEM, var String itemId,
        var String sector, var String itemType,
        var Double x, var Double y)@self

    val HALF_DOWN = new ArmTrajectory(...)
    val COMPLETE_DOWN = new ArmTrajectory(...)
    val UP = new ArmTrajectory(...)
    val ROTATE = new ArmTrajectory(...)
    val LAY_DOWN = new ArmTrajectory(...)
    val INITIAL_POSITION = new ArmTrajectory(...)
    val CLOSE = new GripperTrajectory(...)
    val OPEN = new GripperTrajectory(...)

    eval(new MoveArm(HALF_DOWN))@self
    in(MOVE_ARM_COMPLETED)@self

    eval(new MoveArm(COMPLETE_DOWN))@self
    in(MOVE_ARM_COMPLETED)@self

    eval(new UseGripper(CLOSE))@self
    in(USE_GRIPPER_COMPLETED)@self

    eval(new MoveArm(UP))@self
    in(MOVE_ARM_COMPLETED)@self

    out(ITEM_READY_FOR_DELIVERY,sector)@self

    eval(new MoveArm(ROTATE))@self
    in(MOVE_ARM_COMPLETED)@self

    in(DELIVERY_ROBOT_ARRIVED)@self

    eval(new MoveArm(LAY_DOWN))@self
    in(MOVE_ARM_COMPLETED)@self

    eval(new UseGripper(OPEN))@self
    in(USE_GRIPPER_COMPLETED)@self

    out(GRIPPER_OPENED,itemId,itemType)@self

    eval(new MoveArm(INITIAL_POSITION))@self
    in(MOVE_ARM_COMPLETED)@self

    out(IS_IN_THE_INITIAL_POSITION)@self
}
```

Fig. 10 The process with the logic of the arm robot

```
proc MoveArm(ArmTrajectory armTrajectory) {
    val bridge = new XklaimToRosConnection
        (ROS_BRIDGE_SOCKET_URI)
    val pub = new Publisher("/arm_controller/command",
        "trajectory_msgs/JointTrajectory", bridge)
    val JointTrajectory trajectory = new JointTrajectory()
        .positions(armTrajectory.trajectoryPoints).jointNames(#[
        "joint1","joint2","joint3","joint4","joint5","joint6"])
    pub.publish(trajectory)

    bridge.subscribe(
        SubscriptionRequestMsg
            .generate("/arm_controller/state")
            .setType(
            "control_msgs/JointTrajectoryControllerState")
            .setThrottleRate(1).setQueueLength(1),
        [ data, stringRep |
            val actual = data.get("msg")
                .get("actual").get("positions")

            var delta = 0.0
            for (var i = 0;
                    i < armTrajectory.trajectoryPoints.size; i++)
                delta += Math.pow(actual.get(i).asDouble() -
                    armTrajectory.trajectoryPoints.get(i), 2.0)
            val norm = Math.sqrt(delta)

            if (norm <= armTrajectory.tolerance) {
                out(MOVE_ARM_COMPLETED)@self
                bridge.unsubscribe("/arm_controller/state")
            }
        ]
    )
}
```

Fig. 11 The process for moving the robotic arm

putting a particular tuple which is consumed by the process `PickAndReleaseOneItem` before starting the execution of the next process. On its termination, the process outputs the tuple with `IS_IN_THE_INITIAL_POSITION`, expected by `ArmBehavior` (Fig. 9), to notify that it has finished its tasks.

In Fig. 11, we show the code of the process `MoveArm`. Like `UseGripper` (which we do not show here), `MoveArm` relies on the ROS Bridge. As already discussed in Sect. 3, the execution of an X-KLAIM robotics application requires the ROS Bridge server to run, providing a WebSocket connection at a given URI. The URI of the ROS Bridge WebSocket is one of the Java constants we defined. In the code of our example application, we consider the ROS Bridge server running on the local machine (0.0.0.0) at port 9090.

The process `MoveArm` connects to the ROS Bridge and initializes a publisher for the topic related to the control of arm movements. The process defines the trajectory for the arm movement and publishes it. Then, the process uses the Java API provided by *java_rosbridge* for subscribing to a specific topic (we refer to *java_rosbridge* documentation for the used API). The last argument of `bridge.subscribe` is a lambda expression (see Sect. 2.2). The lambda expression will be executed when an event for the subscribed topic is received. In particular, the lambda expression reads some data from the event (in JSON format) concerning the "positions." ROS dictates the JSON message format. To access the contents, we use the standard Java API (`data` is of type `JsonNode`, from the *jackson-databind* library). The lambda expression calculates the delta between the actual joint positions and the destination positions to measure the arm movement's completeness. The `if` determines when the arm has completed the rotation movement according to a specific tolerance. When that happens, the lambda expression notifies that the task is completed. This is achieved by inserting a particular tuple in the local tuple space. The process in Fig. 10 will consume this tuple and will go on by spawning the process for the next movement. Finally, we can unsubscribe from the topic to stop receiving notifications from the ROS Bridge.

Note that the thread executing `MoveArm` terminates immediately after executing the `bridge.subscribe` call. On the contrary, from a logical point of view, the task of the process `MoveArm` terminates only after the lambda (executed by a different thread, which is part of the *java_rosbridge* publish/subscribe mechanism) has published the tuple `MOVE_ARM_COMPLETED`. This is typical of the asynchronous multi-threaded nature of publish/subscribe frameworks. This is the reason why, to start execution of the next process, the process `PickAndReleaseOneItem` cannot simply wait for the termination of `MoveArm` but leverages the coordination mechanism provided by the tuple space.

```
proc DeliveryOneItem(String robotId,
        String sector, Locality Arm) {
    in(ITEM_READY_FOR_DELIVERY,sector)@Arm

    // the arm robot has a fixed, known position
    val x = −0.21
    val y = 0.31
    eval(new MoveTo(robotId, x, y))@self
    in(MOVE_TO_COMPLETED)@self

    out(DELIVERY_ROBOT_ARRIVED)@Arm

    in(GRIPPER_OPENED,
        var String itemId, var String itemType)@Arm

    eval(new WaitForItem(robotId))@self
    in(ITEM_LOADED)@self

    // the destination has a fixed, known position
    val x2 = −8.0
    val y2 = 0.0
    eval(new MoveTo(robotId, x2, y2))@self
    in(MOVE_TO_COMPLETED)@self

    out(ITEM_DELIVERED,itemId,x2,y2)@self

    out(AVAILABLE_FOR_DELIVERY)@self
}
```

Fig. 12 The process with the logic of the delivery robot

The implementation of the delivery robot (see the node `DeliveryRobot` in Fig. 8) follows a similar strategy. The `DeliveryRobotBehavior` is similar to the behavior of Fig. 9, and we do not show it here.

The process with the logic of the delivery robot is shown in Fig. 12. Similarly to `PickAndReleaseOneItem` of Fig. 10, this process delegates the physical actions to reusable processes that use the ROS Bridge: `WaitForItem` (which we do not show here) and `MoveTo`, which is parameterized over the target destination.

We show `MoveTo` in Fig. 13, including the parts that deal with mathematical computations concerning the currently read position. The parts for using the ROS Bridge and coordinating through the tuple space are similar to the ones of `MoveArm` of Fig. 11. The delivery robot navigation in this process is based on a proportional control technique that adjusts the robot's linear and angular velocities depending on its current position and orientation relative to the target. It calculates the heading error, which is the difference between the angle the robot is currently facing and the angle it needs to reach the target, as well as the distance error. The linear and angular velocities are then adjusted based on proportional gain, where the distance error is proportional to the linear velocity and the heading error is proportional to the angular velocity. This approach is simple but effective for navigating to a specific point in an environment free of obstacles. However, it may not be suitable for more complex scenarios that include static and dynamic objects. In those cases, a more advanced navigation system, like *2D navigation stack* provided by ROS, may be required, as shown in Sect. 4.2.

```
proc MoveTo(String robotId, Double x, Double y) {
    val bridge = new XklaimToRosConnection
        (ROS_BRIDGE_SOCKET_URI)
    val pub = new Publisher("/" + robotId + "/cmd_vel",
        "geometry_msgs/Twist", bridge)
    // set the tolerance for distance and angle error
    val distanceTolerance = 0.1
    val angleTolerance = 0.1
    // create the message for sending velocity commands
    val vel_msg = new Twist()
    val PI = Math.PI
    // gain K used to calculate the linear velocity
    val double K_l = 0.5
    // gain K used to calculate the angular velocity
    val double K_a = 0.5
    bridge.subscribe(
        SubscriptionRequestMsg
            .generate("/" + robotId + "/odom")
            .setType("nav_msgs/Odometry").setThrottleRate(1)
            .setQueueLength(1),
        [ data, stringRep |
            // extract position from the odometry sensor
            var mapper = new ObjectMapper()
            var JsonNode rosMsgNode = data.get("msg")
            var odom = mapper
                .treeToValue(rosMsgNode, Odometry)
                .pose.pose
            var currentX = odom.position.x
            var currentY = odom.position.y
            var angle = new EulerAngles(odom.orientation)
            var currentTheta = angle.yaw
            // calculate the error in heading and distance
            var deltaX = x − currentX
            var deltaY = y − currentY
            // calculating the angle error
            var angular = Math.atan2(deltaY, deltaX)
            var headingError = angular − currentTheta
            if (headingError > PI) {
                headingError = headingError − (2 * PI)
            }
            if (headingError < −PI) {
                headingError = headingError + (2 * PI)
            }
            // calculate the distance to the destination
            var distance = Math.sqrt(
                Math.pow((x − currentX), 2)
                + Math.pow((y − currentY), 2))
            if (distance > distanceTolerance) {
                // move toward the destination
                if (Math.abs(headingError) > angleTolerance) {
                    vel_msg.linear.x = 0.0;
                    vel_msg.angular.z = K_a * headingError
                } else {
                    vel_msg.linear.x = K_l * distance
                    vel_msg.angular.z = 0.0;
                }
                pub.publish(vel_msg)
            } else {
                // the robot reached the goal and stops moving
                vel_msg.linear.x = 0
                vel_msg.angular.z = 0
                pub.publish(vel_msg)
                out(MOVE_TO_COMPLETED)@self
                bridge.unsubscribe("/" + robotId + "/odom")
            }
        ]
    )
}
```

Fig. 13 The process for moving the delivery robot

To recap, we propose an approach that clearly separates the responsibilities among different processes, which can be seen as different logical layers:

- We have a process for the high-level behavior of the robot, like `ArmBehavior` of Fig. 9, which only takes care of establishing the lifecycle of the robot.
- We have a process for implementing the main logic of the robot, like `PickAndReleaseOneItem` of Fig. 10, which relies on the reusable building blocks of parameterized

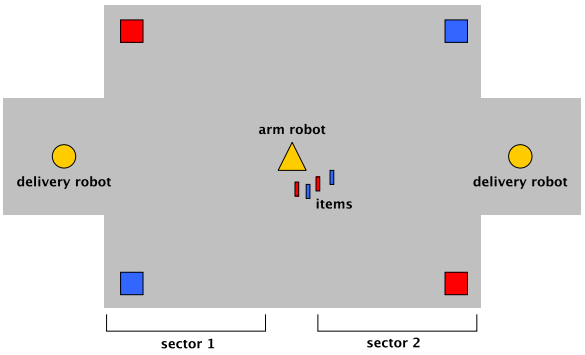

Fig. 14 Enriched warehouse scenario

processes that are responsible for using the ROS Bridge for communicating with the robot's physical parts.
- These latter processes implement the main physical actions. Still, they are reusable thanks to their parameterization. For example, we have a single process representing the "movement of the arm"; according to the parameter, the process will go down, go up, rotate, etc. This is quite different from the approach presented in [12], where we used a different X-KLAIM process for every single movement, e.g., `GetDown`, `GetUp`, `Rotate`, etc. This led to many processes with some code duplicated across those processes, which might become hard to read and maintain.

We believe this clear separation of responsibilities enhances our code's reusability, readability, and maintainability. This also allows us to follow an incremental approach, as shown in this section: we start focusing on a smaller problem/scenario (one single delivery robot, one single item type, no obstacles); then, we scale to a more complex one (two delivery robots, different item types, presence of obstacles), by reusing most of the code of the simple scenario and extending/modifying only a few processes for the goal of the advanced scenario, as we show in the next Sect. 4.2.

4.2 Enriching the warehouse scenario

In this section, we present an evolution of the simple scenario of Sect. 4.1. Most of the code is the same as in Sect. 4.1, and in this section, we focus on the parts that must be adapted for the evolved scenario.

As shown in Fig. 14, in this scenario, the MRS is composed of an arm robot and two delivery robots, and the warehouse is divided into two sectors, each one served by a delivery robot. Similarly to the previous scenario, the arm robot, positioned in the center of the warehouse, picks up one item at a time from the ground, calls the delivery robot assigned to the item's sector, and releases the item on top of the delivery robot. The latter delivers the item to the appropriate delivery area, which depends on the item's color, and then becomes

```
net MRS physical "localhost:9999" {
    node Arm {
        eval(new ArmBehavior())@self
    }

    node DeliveryRobot1 {
        val robotId = "robot1"
        val sector ="sector1"
        out(ITEM_DESTINATION,"red",−9.0,−9.0)@self
        out(ITEM_DESTINATION,"blue",9.0,−9.0)@self
        eval(new DeliveryRobotBehavior
            (robotId, sector, Arm))@self
    }

    node DeliveryRobot2 {
        val robotId = "robot2"
        val sector ="sector2"
        out(ITEM_DESTINATION,"red",9.0,9.0)@self
        out(ITEM_DESTINATION,"blue",−9.0,9.0)@self
        eval(new DeliveryRobotBehavior
            (robotId, sector, Arm))@self
    }

    node Environment {...}
}
```

Fig. 15 The X-KLAIM net of the enriched warehouse scenario

```
proc DeliveryOneItem(String robotId,
        String sector, Locality Arm) {
    in(ITEM_READY_FOR_DELIVERY,sector)@Arm

    val x = −0.21
    val y = 0.31
    eval(new MoveTo(robotId, x, y))@self
    in(MOVE_TO_COMPLETED)@self

    out(DELIVERY_ROBOT_ARRIVED)@Arm

    in(GRIPPER_OPENED,
        var String itemId, var String itemType)@Arm

    eval(new WaitForItem(robotId))@self
    in(ITEM_LOADED)@self

    // the delivery destination coordinates
    // must be retrieved: they are not known
    read(ITEM_DESTINATION, itemType,
        var Double x2, var Double y2)@self

    eval(new MoveTo(robotId, x2, y2))@self
    in(MOVE_TO_COMPLETED)@self

    out(ITEM_DELIVERED,itemId,x2,y2)@self

    out(AVAILABLE_FOR_DELIVERY)@self
}
```

Fig. 16 The process with the logic of the delivery robot in the enriched warehouse scenario

available for a new delivery. In addition, delivery robots must deal with obstacles (e.g., pallets) that are in the warehouse.

In Fig. 15, we show a part of the network for this scenario implementation. The network is similar to the one of Fig. 8 since we reuse the processes for the behavioral parts. The main difference lies in the fact that, since in this scenario the destination coordinates depend on the item type (i.e., color), before activating the `DeliveryRobotBehavior` processes, the delivery robot nodes insert in the local tuple space the `ITEM_DESTINATION` tuples that define the mapping from item type to destination coordinates.

We have to adapt the process `DeliveryOneItem` as shown in Fig. 16. Comparing the code of this process with

```
proc MoveTo(String robotId, Double x, Double y) {
    val bridge = new XklaimToRosConnection
        (ROS_BRIDGE_SOCKET_URI)
    val pub = new Publisher("/" + robotId +
        "/move_base_simple/goal",
        "geometry_msgs/PoseStamped", bridge)
    // publish the destination position
    val destination = new PoseStamped()
        .headerFrameId("world").posePositionXY(x, y)
        .poseOrientation(1.0)
    pub.publish(destination)

    // waiting until the destination position is reached
    bridge.subscribe(
        SubscriptionRequestMsg
            .generate("/" + robotId + "/amcl_pose")
            .setType("geometry_msgs/PoseWithCovarianceStamped")
            .setThrottleRate(1).setQueueLength(1),
        [ data, stringRep |
            // the actual position from the robot's status
            var mapper = new ObjectMapper()
            var JsonNode rosMsgNode = data.get("msg")
            var current_position = mapper
                .treeToValue(rosMsgNode,
                    PoseWithCovarianceStamped)
                .pose.pose
            // calculate the delta between the actual position
            // and the destination position
            // to measure the completeness of the movement
            val tolerance = 0.16
            var deltaX = Math.abs(
                current_position.position.x -
                destination.pose.position.x)
            var deltaY = Math.abs(
                current_position.position.y -
                destination.pose.position.y)
            if (deltaX <= tolerance && deltaY <= tolerance) {
                val pubvel = new Publisher(
                    "/" + robotId + "/cmd_vel",
                    "geometry_msgs/Twist", bridge)
                val twistMsg = new Twist()
                pubvel.publish(twistMsg)
                out(MOVE_TO_COMPLETED)@self
                bridge.unsubscribe("/" + robotId + "/amcl_pose")
            }
        ]
    )
}
```

Fig. 17 The process for moving the delivery robot in the enriched warehouse scenario

the corresponding one of the first scenario (Fig. 12), we see that it retrieves the destination coordinates at run-time from the local tuple space through the `ITEM_DESTINATION` tuples. Notably, the two robots deliver items of the same type to different destinations.

We also have to adapt the process `MoveTo` as shown in Fig. 17. This process is much simpler in this scenario, even if it relies on a more sophisticated navigation system leveraging the navigation stack packages provided by ROS. Specifically, we exploit the topic `move_base_simple` to communicate with the ROS node of the navigation system, which accepts messages containing the goal coordinates. This node is an integral component of the navigation stack. It is responsible for linking the global planner with the local planner to determine the robot's trajectory while avoiding obstacles. The global planner generates the path as a sequence of waypoints the robot must follow. The local planner generates the low-level plan to move from one waypoint to the next. This mechanism relies on a map of the environment and localization facilities to achieve this. To determine the completion of the robot's movement, the process `MoveTo` subscribes to the topic `amcl_pose`, which provides an estimate of the robot's pose in the given map using the AMCL algorithm.

The other processes for implementing this scenario are the same as the corresponding ones of the simple scenario. As anticipated in Sect. 4.1, having separated responsibilities in reusable processes allowed us to follow an incremental approach: we reused most of the code of the simple scenario, and we had to modify/replace only a few processes according to the requirements of this scenario.

The screenshot in Fig. 18 shows this scenario in execution. On the left, the Eclipse IDE with our X-KLAIM code is shown (see the logged messages on the console). On the right, the Gazebo simulator is shown, visualizing the arm in the center, ready to drop the item on top of the delivery robot's white plate.

5 Experimental evaluation

In this section, we illustrate the experiments we carried out to determine the impact of our approach on MRS performance. The experiments are designed to provide a comparison of time and memory performance of our implementation of the MRS of the warehouse scenarios based on Java code and ROS Bridge against the traditional ROS implementation based on Python code.[6] To this aim, we exploit the warehouse scenario in Sect. 4.2, evaluating the overall execution time in milliseconds and memory consumption for each robot activity while using our solution and the traditional one. To guarantee consistency, the same hardware/software,[7] input, and tasks are used in both environments. To account for variance, we ran the same experiments (i.e., the Java-based and the Python-based code) 30 times and averaged the results.[8]

Time consumption We determined the average completion time for each robot activity in milliseconds using our implementation based on *java_rosbridge* and the Python one based on *rospy*.[9] We discuss here the results of the experiments concerning the arm robot's activities; the ones concerning the delivery robots returned similar results. Each activity uses a publisher for sending the message that starts the enactment of the activity and a subscriber for receiving sensor data.

[6] The Python code and the data of all the experiments are available at https://github.com/LorenzoBettini/xklaim-ros-warehouse-scenarios/tree/master/experiments.

[7] We conducted our experiments on a workstation with Intel(R) Core(TM) i7-7700HQ (8 cores, 2.80 GHz) and 32 GB RAM, running Linux Ubuntu 20.04.5 LTS, ROS Noetic, and OpenJDK 64-Bit Server VM 11.0.17.

[8] We use averaged results because the standard deviation in these experiments is low (data of the experimental results are available at https://github.com/LorenzoBettini/xklaim-ros-warehouse-scenarios/tree/master/experiments/Results).

[9] http://wiki.ros.org/rospy.

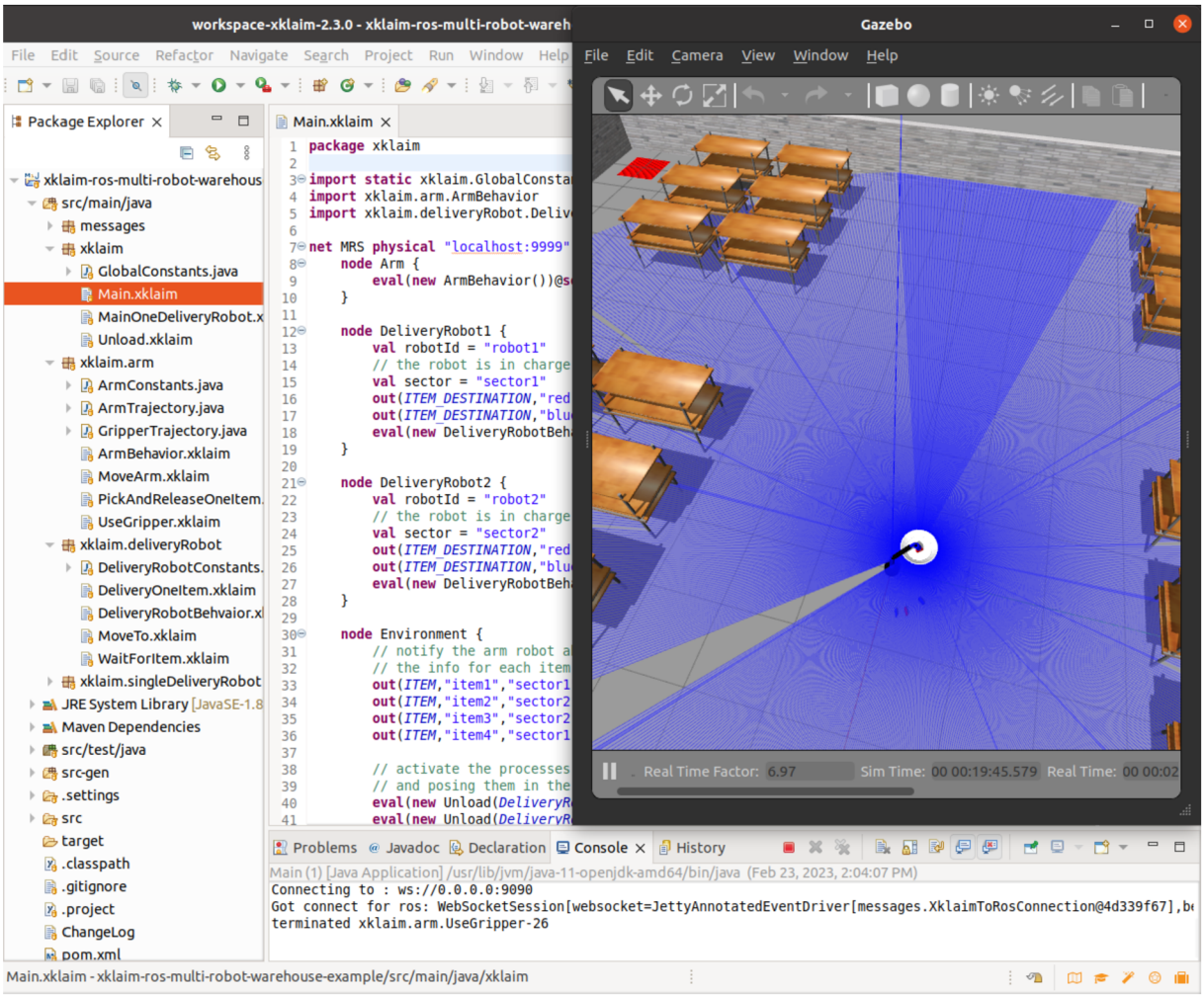

Fig. 18 Execution of the X-KLAIM robotics application of the enriched warehouse scenario

Figure 19 shows the time consumption for the activities of the arm robot. Each activity corresponds to an X-KLAIM process in our approach (except for "Move down," which corresponds to two executions of `MoveArm`, with arguments `HALF_DOWN` and `COMPLETE_DOWN`, respectively) and a class in the Python implementation. For example, the "Rotate" activity takes an average of 3955 milliseconds in Python and 4019 milliseconds in Java to execute. The time consumption is similar for activities using different topics; e.g., "Open gripper" takes 3986 milliseconds in Python and 4021 milliseconds in Java. The experimental results indicate that the Java program has a slightly greater latency than the Python version. This is a consequence of the serialization and deserialization of messages, network overhead, and connection with the ROS Bridge server via the WebSocket protocol. However, in the case under evaluation, the average delay difference between the two setups is at most 200 milliseconds. Therefore, the overhead introduced by our solution does not significantly affect the mission of the considered MRS.

Memory consumption To measure the memory consumption, we employed two different tools for performance profiling and monitoring: VisualVm[10] for Java code and Memray[11] for Python.

We show here the results of the experiments on one robot's activity, namely "Rotate." As shown in Fig. 20, after the start-up phase of the activity, the heap used by Python is more than 50 MB, while Java uses almost 30 MB. The trend for the other activities is quite similar. Even if identifying the cause of this difference is not relevant to our investigation, we think that, in this experiment, the difference might be attributed to more efficient automatic memory management in Java compared to Python. It is worth noting that our approach also requires the execution of the ROS Bridge server, which uses around 135 KB of memory and, hence, does not significantly affect the overall memory cost.

[10] https://visualvm.github.io.

[11] https://github.com/bloomberg/memray.

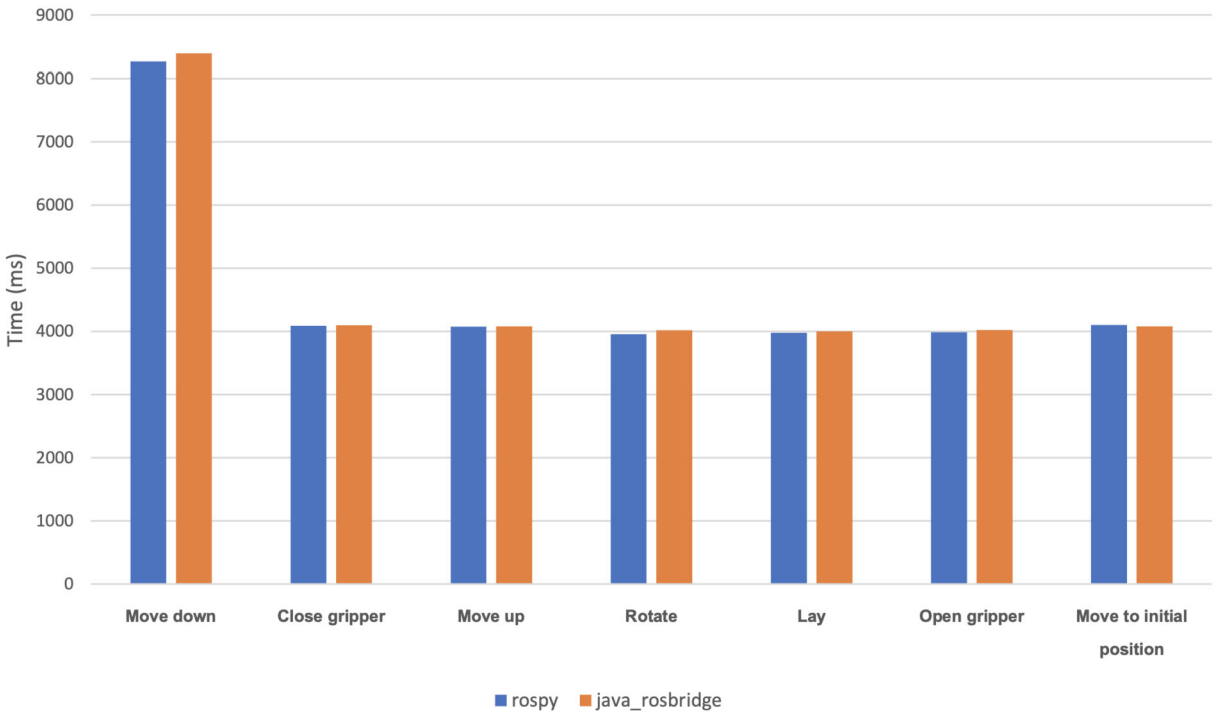

Fig. 19 Time consumption

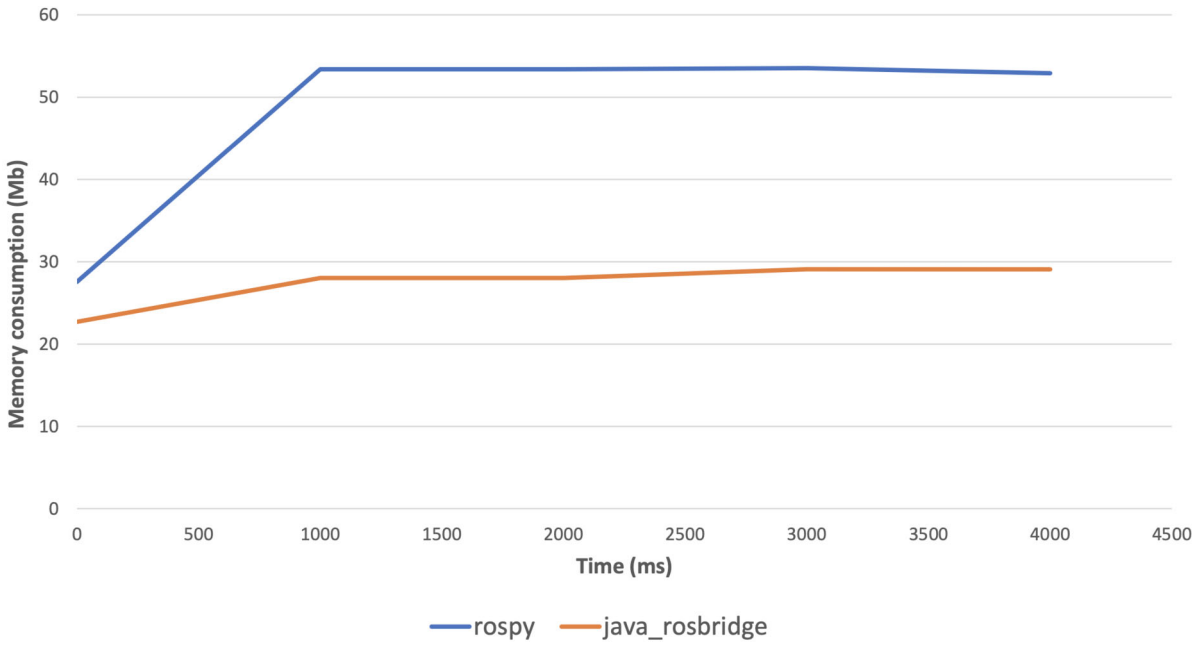

Fig. 20 Memory consumption of the Rotate activity

6 Discussion and related work

Over the last few years, researchers have attempted to define notations closer to the robotics domain to raise the abstraction level for enabling automated code generation, behavior analysis, and property verification (e.g., safety and performance). This section reviews several high-level languages and frameworks for modeling, designing, and verifying ROS-based applications and some languages for coordinating collaborative MRSs. We summarize in Table 1 our considerations and comparison with the languages more strictly related to ours.

Table 1 Features comparison of the related works

DSL	*Formal language*	*High-level language*	*Multi-robots*	*Heterogenous robots*	*Coordination*	*Decentralized coordination*	*Open-endedness*	*Compiler*	*IDE*	*ROS*
ART2ool [23]		✓						✓		✓
ATLAS [24]		✓	✓		✓			✓		✓
BRIDE [25]		✓						✓	✓	✓
CommonLang [26]		✓						✓		✓
Drona [27]	✓	✓	✓		✓	✓		✓		✓
FLYAQ [28]		✓	✓							✓
Hyperflex [29]		✓						✓	✓	✓
ISPL [30]	✓	✓	✓		✓	✓		✓		
Koord [31]	✓	✓	✓	✓	✓	✓		✓		✓
PROMISE [32]		✓	✓							✓
RobotChart [33]	✓	✓						✓	✓	
ROSBuzz [34]	✓	✓	✓	✓	✓	✓		✓	✓	✓
RSSM [35]	✓							✓		✓
SCEL [36]	✓	✓	✓	✓	✓	✓	✓	✓		
X-Klaim	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

High-level languages and frameworks Many DSLs for component-based modeling of robotic systems are based on UML and target mostly the architectural aspect of robotic applications, e.g., RobotML [1], V^{3}CMM [37], BRICS [38], RoboChart [33], and SafeRobots [39]. Some of them can be used to build ROS-based systems by either supporting a direct translation, e.g., Hyperflex [29], or serving as a base for other platforms. For example, in BRIDE [25], which relies on BRICS, the components are modeled using UML and converted to ROS meta-models to generate runnable ROS C++ code. Additional meta-models (i.e., deployment meta-model and experiment meta-model) for rapid prototyping of component-based systems are provided in [40]. UML has also been used to model and design robotic tasks and missions; e.g., Art2ool [23] supports the development cycle of robotic arm tasks in which atomic tasks are abstracted with UML class diagrams. Textual languages, e.g., CommonLang [26], are another type of language used to model robotic systems. For example, in [41], a DSL based on the Python language can be used interactively, through the Python command-line interface, to create brand new ROS nodes and reshape existing ROS nodes by wrapping their communication interfaces.

Some other contributions, to some extent, allow for the verification of ROS-based systems. ROSGen [42] takes a specification of a ROS system architecture as an input and generates ROS nodes as an output. Using the theorem prover Coq, the generation process is amenable to formal verification. DeROS [43] permits describing robots' safety rules (and their related corrective actions) and automatically generating a ROS safety monitoring node by integrating these rules with a run-time monitor. Another framework for run-time verification of ROS-based systems is described in [44], which allows generating C++ code for a monitoring node from user-defined properties specified in terms of event sequences. In [45], robot systems are modeled as a network of timed automata that, after verification in Uppaal,[12] are automatically translated into executable C++ code satisfying the same temporal logic properties as the model. Finally, RSSM [35] enables modeling of multi-agent robots' activities using hierarchical Petri nets. After checking for deadlock absence on the model, RSSM can automatically generate C++ code for ROS packages.

The approaches mentioned above have not been applied to such complex systems as MRSs, and some are not even suitable for such systems. Very few high-level languages for MRSs have been proposed. For example, FLYAQ [28] is a set of DSLs based on UML to specify civilian missions for unmanned aerial vehicles. This work is extended in [46] to enable the use of a declarative specification style, but it only supports homogeneous robots. ATLAS [24], which also provides a simulator-based analysis, takes a step further towards coordinating MRSs but only supports centralized coordination. PROMISE [32] allows specifying the missions of MRSs using linear temporal logic operators for composing robotic mission patterns. Finally, RMoM [47] allows first using a high-level language for specifying various constraints and properties of ROS-based robot swarms with temporal and timed requirements and then automatically generating distributed monitors for their run-time verification.

[12] https://uppaal.org/.

Languages for coordination Coordination for MRSs has been investigated from several diverse perspectives. Nowadays, many techniques can be used to orchestrate the actions and movements of robots operating in the same environment [4, 48]. Designing fully automated and robust MRSs requires strong coordination of the involved robots for autonomous decision making and mission continuity in the presence of communication failures [49]. Several studies recommend using indirect communication to cut implementation and design costs usually caused by direct communication. Indirect communication occurs through a shared communication structure that each robot can access in a distributed concurrent fashion. Some languages providing communication and coordination primitives suitable for designing robust MRSs are reviewed in [5]. In ISPL [30], communication is obtained as an indirect result of synchronizing multiple labeled transition systems on a specific action. In SCEL [36], a formal language for the description and verification of collective adaptive systems, communication is related to the concept of knowledge repositories, represented by tuple spaces. In Buzz [50], a language for programming heterogeneous robot swarms, communication is implemented as a distributed key-value store. For this latter language, integration with the standard environment of ROS has also been developed, which is named Rosbuzz [34]. Unlike X-KLAIM, however, Rosbuzz does not provide high-level coordination primitives, robots' distribution is not explicit, and it permits less heterogeneity. Drona [27] is a framework for distributed drones where communication is somehow similar to the one used in ISPL. Koord [31] is a language for programming and verifying distributed robotic applications where communication occurs through a distributed shared memory. Unlike X-KLAIM, however, robot distribution is not explicit, and open-endedness is not supported. Finally, in [51], a programming model and a typing discipline for complex multi-robot coordination are presented. The programming model uses choreographies to compositionally specify and statically verify message-based communications and jointly executed motion between robotics components in the physical space. Well-typed programs, which are terms of a process calculus, are then compiled into programs in the ROS framework.

7 Concluding remarks and future work

In this paper, we have presented an approach based on the language X-KLAIM and the ROS framework for programming robotics applications involving multiple heterogeneous robots. X-KLAIM has proved expressive enough to smoothly implement MRSs' behaviors, and its integration with Java allowed us to seamlessly use the *java_rosbridge* API directly in the X-KLAIM code to access the publish/subscribe communication infrastructure of ROS. Our experimental results show that the use of X-KLAIM and *java_rosbridge* introduces just a slightly greater but acceptable latency than the traditional ROS implementation based on Python code.

We believe the X-KLAIM computation and communication model is particularly suitable for programming MRSs' behavior. On the one hand, X-KLAIM natively supports concurrent programming, which is required by the distributed nature of robot software. On the other hand, the organization of an X-KLAIM application in terms of a network of nodes interacting via multiple distributed tuple spaces, where communicating processes are decoupled in both space and time, naturally reflects the distributed structure of an MRS. In addition, X-KLAIM tuples permit to model both raw data produced by sensors and aggregated information obtained from such data. This allows programmers to specify the robot's behavior at different levels of granularity, thus permitting to structure the code in logical layers that provide a systematic approach to program MRS missions. Moreover, the form of communication offered by tuple spaces, supported by X-KLAIM, favors the scalability of MRSs in terms of the number of components and robots that can be dynamically added and meets the open-endedness requirement (i.e., robots can dynamically enter or leave the system). Both features are crucial in MRSs.

Our long-term goal is to design a DSL for the robotics domain that, besides being used for automatically generating executable code, is integrated with tools supporting formal verification and analysis techniques. These tools are indeed highly desirable for such complex and often safety-critical systems as autonomous robots [52]. The tools already developed for KLAIM, e.g., type systems [53–56], behavioral equivalences [57], flow logic [58], and model checking [59–61], could be a valuable starting point. A first attempt to define a formal verification approach for the design of MRSs using the KLAIM stochastic extension StoKlaim and the relative stochastic logic MoSL [60] has been presented in [62]. Along this direction, we plan to investigate the integration of the proposed approach with spatial model checking [63], as done in [64] for a monitoring scenario involving agents moving in physical space. For example, this would permit to guarantee that the robots do not cross unauthorized zones without first signaling themselves in some authorization area or to verify whether all items are reachable without crossing a given zone. In addition, as the completion time of the robots' activities may be crucial in some robotics scenarios, we also intend to consider the analysis of spatiotemporal properties, as in [65].

Run-time adaptation is another important capability of MRSs. In [66], we have shown that adaptive behaviors can be smoothly rendered in KLAIM by exploiting tuple-based higher-order communication to exchange code and possibly execute it. We plan to investigate to what extent we can

benefit from this mechanism to achieve adaptive behaviors in robotics applications. For example, an X-KLAIM process (a controller or an actuator) could dynamically receive code from other possibly distributed processes containing the logic to continue the execution.

X-KLAIM has several other features that we did not use in this work. We list here the most interesting ones, which could be useful for future work in the field of MRSs. Non-blocking versions of `in` and `read` are available: `in_nb` and `read_nb`, respectively. These are useful for checking the presence of a matching tuple without being blocked indefinitely. In that respect, X-KLAIM also provides "timed" versions of these operations: as an additional argument, they take a timeout, which specifies how long the process executing such action is willing to wait for a matching tuple. If a matching tuple is not found within the specified timeout, the programmer can adopt adequate countermeasures. In the example of this paper, we used the simplest way of specifying a *flat* and *closed* network in X-KLAIM. However, X-KLAIM also implements the hierarchical version of the KLAIM model as presented in [21], which allows nodes and processes to be dynamically added to existing networks so that modular programming can be achieved and *open-ended* scenarios can be implemented.

It is worth noting that in this work, we exploit both the tuple-based communication model, which X-KLAIM inherits from KLAIM, and the publish/ subscribe one, supported by ROS and enabled in X-KLAIM by the *java_rosbridge* library. The former communication model is used to coordinate both the execution of concurrent processes running in a robot and the inter-robot interactions. The latter model, instead, is used to send/receive messages for given topics to/from the ROS framework installed in a single robot. In principle, the former model can be used to express the latter. However, this would require introducing intermediary processes that consume tuples and publish their data on the related topics and, vice versa, generate a tuple each time an event for a subscribed topic is received. This would introduce significant overhead in the communication with the ROS framework, especially for what concerns the handling of the subscriptions (as topics related to sensors usually produce message streams). In this work, we have shown how the use of the publish/subscribe mechanism can be made transparent to the programmer, overcoming the performance issue by elevating the level of abstraction. The programming framework we provide does not replace topics with tuples but offers ready-to-use reusable processes acting as building blocks for creating robotics applications. These processes will hide the interactions with ROS to the programmer and produce tuples only when events relevant to the coordination of the MRS behavior occur (e.g., a robot has reached a given position or a requested movement has been completed). For example, the `MoveArm` process performs different movements of the robot's arm depending on the argument passed when the process is called. It notifies the completion of the movement by emitting a given tuple in the local tuple space.

MRSs act in highly dynamic and uncertain environments, which may lead such systems to face unpredictable or not fully codified situations. In these cases, an advanced decision support system empowered with AI technology can be helpful in deciding the action to take. It is possible to integrate AI functionalities in an X-KLAIM application at different levels in different ways:

- By using an existing ROS package that provides AI functionalities. This solution does not require any development effort and is completely transparent to the X-KLAIM code, which can activate and take advantage of the new functionalities by resorting to the publish/subscribe communication mechanism as usual.
- By using existing Python libraries (e.g., TensorFlow,[13] Keras,[14] PyTorch,[15] scikit-learn[16]) to define custom AI models and exposing them as ROS nodes. Again, once the ROS nodes have been created, this solution is completely transparent to the X-KLAIM code.
- By importing an existing Java library (e.g., Deeplearning4j,[17] DJL[18]) or a Java wrapper of a library written in another language. This way, the AI functionalities will be directly and easily accessible from the X-KLAIM code, thanks to the interoperability with Java provided by XBASE.

We plan to investigate these kinds of integration in future work.

Finally, in this work, we have used version 1 of ROS as a reference middleware for the proposed approach because, currently, this seems to be most adopted in practice. We plan anyway to investigate the possibility of extending our approach to version 2 of ROS, which features a more sophisticated publish/subscribe system based on the OMG DDS standard.

Acknowledgements We thank the anonymous referees for their useful comments.

Funding Open access funding provided by Università degli Studi di Firenze within the CRUI-CARE Agreement. This work was partially supported by the PRIN projects "SEDUCE" n. 2017TWRCNB and "T-LADIES" n. 2020TL3X8X, the INdAM - GNCS Project "Proprietà qualitative e quantitative di sistemi reversibili" n. CUP_E55F2200027001, and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NextGenerationEU.

[13] https://www.tensorflow.org.

[14] https://keras.io/.

[15] https://pytorch.org/.

[16] https://scikit-learn.org/.

[17] https://deeplearning4j.konduit.ai/.

[18] https://djl.ai/.

Declarations

Competing Interests The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. All authors contributed equally to this work.

References

1. Dhouib, S., et al.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Proc. of SIMPAR. LNCS, vol. 7628, pp. 149–160. Springer, Berlin (2012)
2. Frigerio, M., Buchli, J., Caldwell, D.G.: A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In: Proc. of DSLRob'11. CoRR (2013). arXiv:1301.7190
3. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. Softw. Eng. Robot. **7**, 75–99 (2016)
4. Doriya, R., Mishra, S., Gupta, S.: A brief survey and analysis of multi-robot communication and coordination. In: Int. Conf. on Computing, Communication, Automation, pp. 1014–1021 (2015)
5. De Nicola, R., Di Stefano, L., Inverso, O.: Toward formal models and languages for verifiable multi-robot systems. Front. Robot. AI **5**, 94 (2018)
6. Quigley, M., et al.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
7. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: SIMPAR. LNCS, vol. 8810, pp. 195–206. Springer, Berlin (2014)
8. de Araújo Silva, E., Valentin, E., Carvalho, J.R.H., da Silva Barreto, R.: A survey of model driven engineering in robotics. Comput. Lang. **62**, 101021 (2021)
9. Casalaro, G.L., et al.: Model-driven engineering for mobile robotic systems: a systematic mapping study. Softw. Syst. Model. (2021)
10. Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Writing robotics applications with X-Klaim. In: ISoLA 2020. LNCS, vol. 12477, pp. 361–379. Springer, Heidelberg (2020)
11. Bettini, L., Merelli, E., Tiezzi, F.: X-Klaim is back. In: Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 115–135. Springer, Berlin (2019)
12. Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Programming multi-robot systems with X-Klaim. In: Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning. LNCS, vol. 13703, pp. 283–300. Springer, Berlin (2022)
13. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng. **24**(5), 315–330 (1998)
14. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice Hall, New York (1989)
15. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
16. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java package for distributed and mobile applications. Softw. Pract. Exp. **32**(14), 1365–1394 (2002)
17. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A flexible and modular framework for implementing infrastructures for global computing. In: DAIS. LNCS, vol. 3543, pp. 181–193. Springer, Berlin (2005)
18. Bettini, L., De Nicola, R., Pugliese, R., Ferrari, G.L.: Interactive mobile agents in X-Klaim. In: WETICE, pp. 110–117. IEEE Computer Society, Los Alamitos (1998)
19. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing (2016)
20. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: GPCE, pp. 112–121. ACM, New York (2012)
21. Bettini, L., Loreti, M., Pugliese, R.: An infrastructure language for open nets. In: SAC, pp. 373–377. ACM, New York (2002)
22. Koenig, N.P., Howard, A.: Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: IROS, pp. 2149–2154. IEEE Press, New York (2004)
23. Estévez, E., et al.: ART2ool: a model-driven framework to generate target code for robot handling tasks. Adv. Manuf. Technol. **97**(1–4), 1195–1207 (2018)
24. Harbin, J., et al.: Model-driven simulation-based analysis for multi-robot systems. In: 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS) (2021)
25. Bubeck, A., et al.: BRIDE - a toolchain for framework-independent development of industrial service robot applications. In: ISR, pp. 137–142. VDE, (2014)
26. Rutle, A., Backer, J., Foldøy, K., Bye, R.T.: CommonLang: A DSL for defining robot tasks. In: Proc. of MODELS18 Workshops. CEUR Workshop Proc., vol. 2245, pp. 433–442 (2018)
27. Desai, A., Saha, I., Yang, J., Qadeer, S., Seshia, S.A.: Drona: a framework for safe distributed mobile robotics. In: 8th Intern. Conference on Cyber-Physical Systems, pp. 239–248 (2017)
28. Ciccozzi, F., et al.: Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems. IEEE Access **4**, 6451–6466 (2016)
29. Brugali, D., Gherardi, L.: Hyperflex: a model driven toolchain for designing and configuring software control systems for autonomous robots. In: Robot Operating System. Studies in Computational Intelligence, vol. 625, pp. 509–534. Springer, Berlin (2016)
30. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas: an open-source model checker for the verification of multi-agent systems. Softw. Tools Technol. Transf. **19**(1), 9–30 (2017)
31. Ghosh, R., et al.: Koord: a language for programming and verifying distributed robotics application. Proc. ACM Program. Lang. **4**(OOPSLA), 1–30 (2020)
32. García, S., et al.: High-level mission specification for multiple robots. In: 12th ACM SIGPLAN Int. Conf on Software Language Engineering, pp. 127–140 (2019)
33. Miyazawa, A., et al.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. **18**(5), 3097–3149 (2019)
34. St-Onge, D., Varadharajan, V.S., Li, G., Svogor, I., Beltrame, G.: ROS and Buzz: consensus-based behaviors for heterogeneous teams CoRR (2017). arXiv:1710.08843
35. Figat, M., Zieliński, C.: Robotic system specification methodology based on hierarchical Petri nets. IEEE Access **8**, 71617–71627 (2020)
36. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7 (2014)

37. Alonso, D., et al.: V^{3}CMM: a 3-view component meta-model for model-driven robotic software development. J. Softw. Eng. Robot. **1**, 3–17 (2010)
38. Bruyninckx, H., et al.: The BRICS component model: a model-based development paradigm for complex robotics software systems. In: SAC, pp. 1758–1764. ACM, New York (2013)
39. Ramaswamy, A., Monsuez, B., Tapus, A.: SafeRobots: A model-driven approach for designing robotic software architectures. In: Proc. of CTS, pp. 131–134. IEEE, New York (2014)
40. Kumar, P., et al.: ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS. In: Int. Symp. on Rapid System Prototyping (RSP) (2015)
41. Adam, S., Schultz, U.P.: Towards interactive, incremental programming of ROS nodes. In: Workshop on Domain-Specific Languages and Models for Robotic Systems (2014)
42. Meng, W., Park, J., Sokolsky, O., Weirich, S., Lee, I.: Verified ROS-based deployment of platform-independent control systems. In: NASA Formal Methods Symposium, pp. 248–262. Springer, Berlin (2015)
43. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Rule-based dynamic safety monitoring for mobile robots. J. Softw. Eng. Robot. **7**(1), 121–141 (2016)
44. Huang, J., et al.: ROSRV: runtime verification for robots. In: Int. Conf. on Runtime Verification, pp. 247–254. Springer, Berlin (2014)
45. Wang, R., Guan, Y., Song, H., Li, X., Li, X., Shi, Z., Song, X.: A formal model-based design method for robotic systems. IEEE Syst. J. **13**(1), 1096–1107 (2018)
46. Dragule, S., Meyers, B., Pelliccione, P.: A generated property specification language for resilient multirobot missions. In: SERENE. LNCS, vol. 10479, pp. 45–61. Springer, Berlin (2017)
47. Hu, C., Dong, W., Yang, Y., Shi, H., Zhou, G.: Runtime verification on hierarchical properties of ROS-based robot swarms. IEEE Trans. Reliab. **69**(2), 674–689 (2019)
48. Yan, Z., Jouandeau, N., Ali, A.: A survey and analysis of multi-robot coordination. Int. J. Adv. Robot. Syst. **10**, 1 (2013)
49. Farinelli, A., Iocchi, L., Nardi, D.: Multirobot systems: a classification focused on coordination. IEEE Trans. Syst. Man Cybern., Part B, Cybern. **34**(5), 2015–2028 (2004)
50. Pinciroli, C., Lee-Brown, A., Beltrame, G.: A tuple space for data sharing in robot swarms. EAI Endorsed Trans. Collab. Comput. **2**(9), 2 (2016)
51. Majumdar, R., Yoshida, N., Zufferey, D.: Multiparty motion coordination: from choreographies to robotics programs. Proc. ACM Program. Lang. **4**(OOPSLA), 134 (2020)
52. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems. ACM Comput. Surv. **52**, 1–41 (2020)
53. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. Theor. Comput. Sci. **240**(1), 215–254 (2000)
54. Gorla, D., Pugliese, R.: Enforcing security policies via types. In: SPC. LNCS, vol. 2802, pp. 86–100. Springer, Berlin (2003)
55. Gorla, D., Pugliese, R.: Resource access and mobility control with dynamic privileges acquisition. In: ICALP. LNCS, vol. 2719, pp. 119–132. Springer, Berlin (2003)
56. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. Sci. Comput. Program. **63**(1), 57–87 (2006)
57. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. Inf. Comput. **205**(10), 1491–1525 (2007)
58. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. **75**(6), 376–397 (2010)
59. De Nicola, R., Loreti, M.: A modal logic for mobile agents. ACM Trans. Comput. Log. **5**(1), 79–128 (2004)
60. De Nicola, R., Katoen, J., Latella, D., Loreti, M., Massink, M.: Model checking mobile stochastic logic. Theor. Comput. Sci. **382**(1), 42–70 (2007)
61. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. Sci. Comput. Program. **99**, 24–74 (2015)
62. Gjondrekaj, E., et al.: Towards a formal verification methodology for collective robotic systems. In: ICFEM12. LNCS, vol. 7635, pp. 54–70. Springer, Berlin (2012)
63. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: VoxLogicA: A spatial model checker for declarative image analysis. In: TACAS 2019. LNCS, vol. 11427, pp. 281–298. Springer, Berlin (2019)
64. Basile, D., ter Beek, M.H., Ciancia, V.: An experimental toolchain for strategy synthesis with spatial properties. In: ISoLA 2022. LNCS, vol. 13703, pp. 142–164. Springer, Berlin (2022)
65. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M.: Spatio-temporal model checking of vehicular movement in public transport systems. Int. J. Softw. Tools Technol. Transf. **20**(3), 289–311 (2018)
66. Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F.: Modeling adaptation with a tuple-based coordination language. In: SAC12, pp. 1522–1527. ACM, New York (2012)