Programming Multi-robot Systems with X-KLAIM

Lorenzo Bettini[1], Khalid Bourr[2], Rosario Pugliese[1], and Francesco Tiezzi[1(✉)]

[1] Dipartimento di Statistica, Informatica, Applicazioni, Università degli Studi di Firenze, Firenze, Italy
{lorenzo.bettini,rosario.pugliese,francesco.tiezzi}@unifi.it
[2] School of Science and Technology, Università di Camerino, Camerino, Italy
khalid.bourr@unicam.it

Abstract. Software development for robotics applications is still a major challenge that becomes even more complex when considering a Multi-Robot System (MRS). Such a 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 to coordinate 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 and maintainable code. The proposed approach can be used in practice through the integration of X-KLAIM and the popular robotics framework ROS. We show the proposal's feasibility and effectiveness by implementing an MRS scenario.

Keywords: Multi-robot systems · Multiple tuple spaces · X-KLAIM · ROS

1 Introduction

Autonomous robots are software-intensive systems 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 greater reliability. On the other hand, an MRS requires robots to cooperate and coordinate to achieve common goals.

This work was partially supported by the PRIN projects "SEDUCE" n. 2017TWRCNB and "T-LADIES" n. 2020TL3X8X, and the INdAM - GNCS Project "Proprietà qualitative e quantitative di sistemi reversibili" n. CUP_E55F2200027001.

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13703, pp. 283–300, 2022.
https://doi.org/10.1007/978-3-031-19759-8_18

The development of the software controlling a single autonomous robot is still a challenge [24,31,49]. This becomes even more arduous in the case of MRSs [14, 25], 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 and simplify the rapid prototyping of robotics applications. Among them, nowadays, a prominent solution is ROS (Robot Operating System [52]), 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 a multitude of sensors and actuators, for a large variety of 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 face 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 [11,13,49,50].

Along this line of research, we introduced in [5] an approach for programming a single-robot system. Specifically, we propose using the language X-Klaim [7] 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 this paper, we take a step forward in this direction by extending the approach in [5] to program MRSs. In fact, the X-Klaim's computation and communication model is particularly suitable for dealing both with *(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 both for intra- and inter-robot communication. This simplifies the design and implementation of MRS's software in terms of an X-Klaim application distributed across both multiple threads of execution and multiple hardware platforms, resulting in a better readable, maintainable, and reusable code.

Our framework can be thought of as 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 in complex environments. To illustrate the proposed approach, we consider a warehouse scenario, where an MRS involving an arm robot and two delivery robots manages the movement of items.

The rest of the paper is organized as follows. In Sect. 2, we provide some background notions concerning the X-Klaim language, while in Sect. 3 we present our approach. In Sect. 4 we (partially) illustrate the implementation of a simple robotics scenario according to the proposed approach. In Sect. 5 we present a systematic analysis of the strictly related work, while in Sect. 6 we conclude and touch upon directions for future work.

2 The X-KLAIM Language

This section briefly describes the key ingredient of the approach we propose: the programming language X-KLAIM.[1] We refer the interested reader to the cited sources in the following for a complete account.

X-KLAIM is based on KLAIM (Kernel Language for Agents Interaction and Mobility, [15]), a formal language devised to design distributed applications consisting of (possibly mobile) software components deployed over the nodes of a network infrastructure. KLAIM generalizes the notion of *generative communication*, introduced by the coordination language Linda [33], 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. Tuple spaces are identified through *localities*, which are symbolic addresses of *network nodes* where processes and tuples can be allocated.

Processes can run concurrently, either at the same node or at different nodes, by executing actions to exchange tuples and to move processes. Action **out**(tuple)@nodeLocality adds the specified tuple to the tuple space of the target node identified by nodeLocality. A tuple is a sequence of actual fields, i.e., expressions, localities, or processes. Action **in**(template)@nodeLocality (resp., **read**(template)@nodeLocality) withdraws (resp., reads) tuples from the tuple space hosted at nodeLocality. The process is blocked until a matching tuple is found. *Templates* are sequences of *actual* and *formal* fields, where the latter are used to bind variables to values, localities, or processes. A template matches a tuple if both have the same number of fields and corresponding fields do 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 nodeLocality. A process can use the reserved locality self to refer to its current hosting node.

The implementation of KLAIM consists of the Java package KLAVA (KLAIM in Java [4]), which provides the KLAIM concepts in terms of Java classes and methods, and X-KLAIM (eXtended KLAIM [7]), a Java-like programming language providing KLAIM constructs besides the typical high-level programming constructs. X-KLAIM is translated into Java code that uses the Java package KLAVA. An X-KLAIM program can smoothly access any Java type and Java library available in the project's classpath. X-KLAIM comes with a complete IDE support based on Eclipse. The syntax of X-KLAIM is similar to Java, thus it should be easily understood by Java programmers, but it removes much "syntactic noise" from Java.

3 The X-KLAIM Approach to Multi-robot Programming

In this section, we provide an overview of our approach, and the resulting software framework, for programming MRS applications using ROS and X-KLAIM.

[1] https://github.com/LorenzoBettini/xklaim.

A single autonomous robot has a distributed architecture, consisting of cooperating components, in particular sensors and actuators. Such cooperation is enabled and controlled by the ROS framework,[2] which provides tools and libraries for simplifying the development of complex controllers while abstracting from the underlying hardware. The core element of the ROS framework is the message-passing middleware, which enables hardware abstraction for a wide variety of robotic platforms. Although ROS supports different communication mechanisms, in this paper we only use the most common one: the anonymous and asynchronous publish/subscribe mechanism. For sending a message, a process has to publish it in a *topic*, which is a named and typed bus. A process that is interested in such message has to subscribe to the topic. Whenever a new message is published in the topic, the subscriber will be notified. Multiple publishers and subscribers for the same topic are allowed.

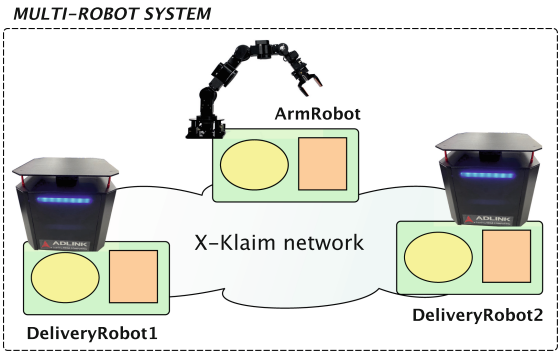

Fig. 1. Software architecture of an MRS in X-KLAIM.

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. 1, 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 local data, i.e., 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 practice, to program the behaviors of the robots forming an MRS, we enabled X-KLAIM programs to interact with robots' physical components by

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

integrating the X-KLAIM language with the ROS middleware. The communication infrastructure of the integrated framework is based on ROS Bridge. This is a server included in the ROS framework that 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. 2 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 $(-0.21, 0.31)$, which is close to the position of the arm robot.

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

Fig. 2. Example of a JSON message for the `/goal` topic.

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*.[3] In its own turn, *java_rosbridge* communicates with the ROS Bridge server, via the WebSocket protocol, by means of the Jetty web server.[4]

ROS permits to check the execution of the code generated from an X-KLAIM program by means of the Gazebo[5] simulator. Gazebo [42] 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 in 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 design activity of the MRS software may benefit from the use of a simulator, to save time and reduce the development cost.

Since the X-KLAIM compiler generates plain Java code, which depends only on KLAVA and a few small libraries, deploying an X-KLAIM application can be done by using standard Java tools and mechanisms. In the context of this paper, 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", and deploy it to a physical robot where a Java virtual machine is already installed. Under 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.

[3] https://github.com/h2r/java_rosbridge.
[4] Jetty 9: https://www.eclipse.org/jetty/.
[5] https://gazebosim.org/.

4 The X-KLAIM Approach at Work on an MRS Scenario

To illustrate the proposed approach, in this section, we show and briefly comment on a few interesting parts of implementing a warehouse scenario[6] involving an MRS that manages the movement of items. As shown in Fig. 3, 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. 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 available for a new delivery.

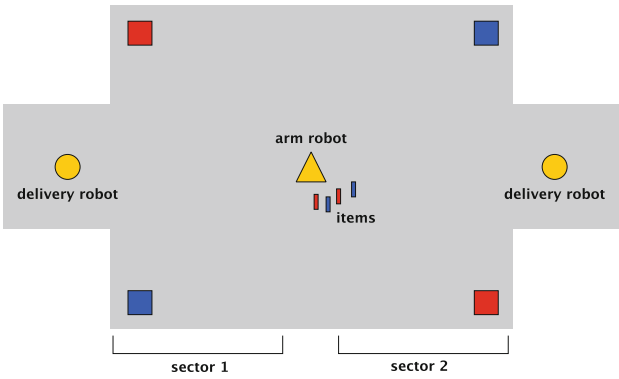

Fig. 3. Warehouse scenario.

In Fig. 4 we show a part of the network for our implementation of the scenario. Each robot is rendered as an X-KLAIM node, whose name represents its locality (see Sect. 2). We have one or several processes for each node implementing the robot's main tasks. Each node creates processes locally and executes them concurrently using the X-KLAIM operation `eval`. Processes are parametric concerning the URI of the ROS bridge WebSocket. 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. 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. Similarly, to execute the code in a simulated environment and obtain a 3D visualization of the execution, the Gazebo simulator has to be launched with the corresponding robot description. At this point, our application can be executed by running the Java class `Main`, which the X-KLAIM compiler has generated. A few processes require additional parameters like the robot and sector id and the locality of other nodes (e.g., the arm's node locality in `MoveToArm`).

[6] The complete source code of the scenario implementation, and a screencast showing its execution on Gazebo, can be found at https://github.com/LorenzoBettini/xklaim-ros-multi-robot-warehouse-example.

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. In the code snippets, we omitted the Java-like import statements. Variable declarations start with `val` or `var`, for final and non-final variables, respectively. The types of variables can be omitted if they can be inferred from the initialization expression. Here we also see the typical KLAIM operations, `read`, `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).

In Fig. 4, the main processes of the nodes wait for specific matching tuples before starting a new loop. To make things simpler, the loop is infinite, but we could easily rely on a termination condition to stop the whole example's net. The main idea behind the implementation of our example is that processes coordinate themselves through the X-KLAIM tuple space-based communication. On the other hand, the processes still rely on the ROS bridge to coordinate the

```
net MRS {
  node Arm {
    val rosbridgeWebsocketURI = "ws://0.0.0.0:9090"
    while (true) {
      in("initialPosition")@self
      in("item", var String itemId, var String sector, var String itemType,
        var Double x, var Double y)@self
      eval(new GetDown(rosbridgeWebsocketURI,x,y))@self
      eval(new Grip(rosbridgeWebsocketURI))@self
      eval(new GetUp(rosbridgeWebsocketURI,x,y))@self
      eval(new Rotate(rosbridgeWebsocketURI,sector))@self
      eval(new Lay(rosbridgeWebsocketURI))@self
      eval(new Release(rosbridgeWebsocketURI,itemId,itemType))@self
      eval(new GoToInitialPosition(rosbridgeWebsocketURI))@self
    }
  }
  node DeliveryRobot1 {
    val rosbridgeWebsocketURI = "ws://0.0.0.0:9090"
    val robotId = "robot1"
    val sector ="sector1"
    while (true) {
      in("availableForDelivery")@self
      eval(new MoveToArm(rosbridgeWebsocketURI,robotId,sector,Arm))@self
      eval(new DeliverItem(rosbridgeWebsocketURI,robotId,Arm))@self
    }
  }
  node DeliveryRobot2 { ... }
  node SimuationHandler { ... }
}
```

Fig. 4. The X-KLAIM net of the warehouse scenario.

physical parts of the robots themselves. This approach can be seen in the code of two of the processes we comment on in this section.

In Fig. 5 we show the code of the process `Rotate`, executed in the node `Arm`. All the processes of this example start by waiting for a specific tuple before executing the main body. This way, the processes that execute in parallel (see the `eval` in Fig. 4) can coordinate themselves: a process will effectively begin its task only after the previous process terminated its task. Then, the process creates the ROS bridge and initializes a publisher for the topic related to the control of the arm movements. After creating the joint positions for the arm movement, the process publishes the trajectory to rotate the arm. The process also inserts a tuple, consisting of an identifier string and the sector, in its local tuple space. The presence of this tuple triggers the call for a delivery robot. In fact, as shown later in Fig. 6, such a tuple is consumed by the `MoveToArm` process, which is executed by the delivery robots. This form of tuple-based interaction between the two kinds of robots allows the arm's code not to depend on the number, the status, and the identities of the delivery robots. This way, the introduction of new delivery robots in the scenario would not affect the code of the arm robot.

```
proc Rotate(String rosbridgeWebsocketURI, String sector) {
  in("getUpCompleted")@self
  val bridge = new XklaimToRosConnection(rosbridgeWebsocketURI)
  val pub = new Publisher("/arm_controller/command",
        "trajectory_msgs/JointTrajectory", bridge)
  val jointPositions = #[−0.9546, −0.20, −0.7241, 3.1400, 1.6613, −0.0142]
  val JointTrajectory rotateTrajectory = new JointTrajectory().positions(jointPositions)
    .jointNames(#["joint1","joint2","joint3","joint4","joint5","joint6"])
  out("itemReadyForTheDelivery",sector)@self
  pub.publish(rotateTrajectory)
  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
      val tolerance = 0.008
      for (var i = 0; i < jointPositions.size; i++)
        delta += Math.pow(actual.get(i).asDouble() − jointPositions.get(i), 2.0)
      val norm = Math.sqrt(delta)
      if (norm <= tolerance) {
        out("rotationCompleted")@self
        bridge.unsubscribe("/arm_controller/state")
      }
    ])
}
```

Fig. 5. The X-KLAIM `Rotate` process.

The process then 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 is a lambda expression (i.e., an anonymous function). In X-KLAIM, *lambda expressions* have the shape `[ param1, param2, ... | body ]`, where the types of the parameters can be omitted if they can be inferred from the context. The lambda will be executed when an event for the subscribed topic is received. In particular, the lambda 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 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 activates the process responsible for raising the object (`GetUp` in our example, see Fig. 4). This is achieved, once again, by inserting a specific tuple in the local tuple space. Finally, we can unsubscribe from the topic so that this process will receive no further notifications from the ROS bridge.

```
proc MoveToArm(String rosbridgeWebsocketURI,String robotId,String sector,Locality arm) {
  val x = −0.21
  val y = 0.31
  in("itemReadyForTheDelivery",sector)@arm
  val bridge = new XklaimToRosConnection(rosbridgeWebsocketURI)
  val pub = new Publisher("/" + robotId + "/move_base_simple/goal",
          "geometry_msgs/PoseStamped", bridge)
  val destination = new PoseStamped().headerFrameId("world")
    .posePositionXY(x, y).poseOrientation(1.0)
  pub.publish(destination)
  bridge.subscribe(
    SubscriptionRequestMsg.generate("/" + robotId + "/amcl_pose").setType(
      "geometry_msgs/PoseWithCovarianceStamped").setThrottleRate(1).setQueueLength(1),
    [ data, stringRep |
      var mapper = new ObjectMapper()
      var rosMsgNode = data.get("msg")
      var current_position = mapper.treeToValue(rosMsgNode, PoseWithCovarianceStamped)
      val tolerance = 0.16
      var deltaX = current_position.pose.pose.position.x − destination.pose.position.x
      var deltaY = current_position.pose.pose.position.y − destination.pose.position.y
      if (deltaX <= tolerance && deltaY <= tolerance) {
        val pubvel = new Publisher("/" + robotId + "/cmd_vel",
          "geometry_msgs/Twist", bridge)
        pubvel.publish(new Twist())
        out("ready")@arm
        out("readyToReceiveTheItem")@self
        bridge.unsubscribe("/" + robotId + "/amcl_pose")
      }
    ])
}
```

Fig. 6. The X-KLAIM `MoveToArm` process.

In Fig. 6 we show the code of the process `MoveToArm`, executed in the node `DeliveryRobot1`. This process is responsible for moving the delivery robot to the arm to get the item deposited on the robot by the arm. The structure of this process is similar to the previous one. Since the arm robot has a fixed position in our scenario, the coordinates `x` and `y` are defined as constants. As anticipated above, this process first waits for a tuple deposited by the `Rotate` process (Fig. 5). Recall that the `Rotate` process deposits such a tuple at its locality, so the process `MoveToArm` retrieves a matching tuple at the locality of the node of the arm (passed as a parameter to the process). The process then publishes the destination position on the ROS bridge and waits until the destination is reached by subscribing to a specific topic. As before, we specify a lambda that decides when the destination has been reached. Also in this case, we use the published information as JSON messages. Once the lambda establishes that the delivery robot arrived at the arm robot's desired position, it stops the wheels (by publishing a `Twist` message). Then, it notifies the arm robot that the delivery robot is ready to receive the item (that is, the arm can drop the item), again, by inserting a tuple at the arm locality. The other tuple inserted in the local tuple space will be retrieved by the `DeliverItem`

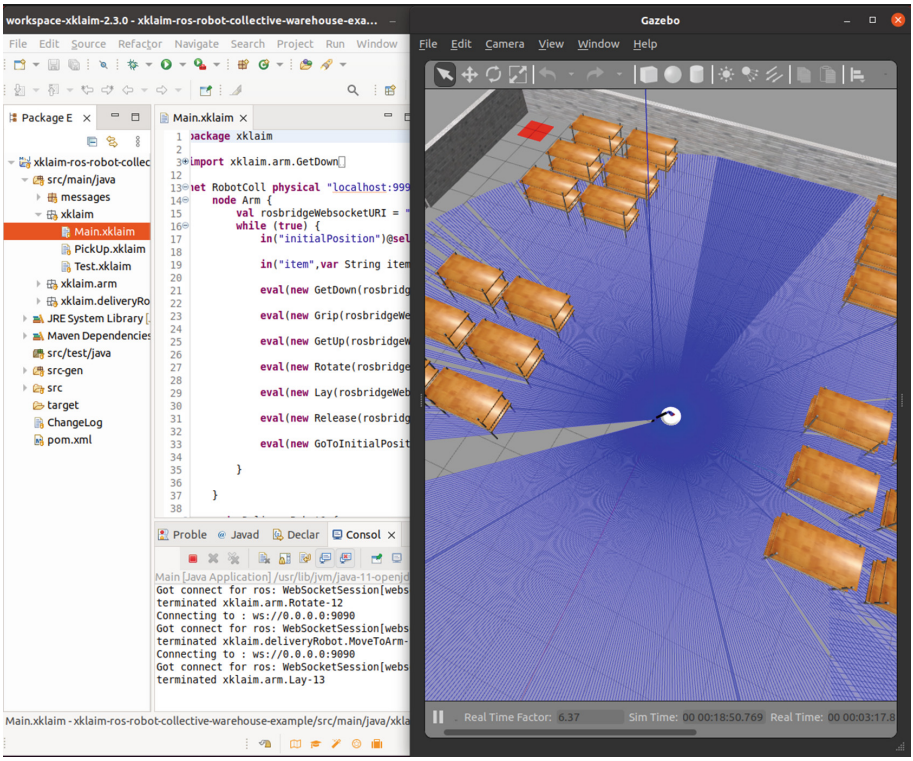

Fig. 7. Execution of an X-KLAIM robotics application.

process, not shown here. As usual, the lambda takes care of unsubscribing from the ROS bridge once done.

The screenshot in Fig. 7 shows our X-KLAIM robotics application 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, which visualizes on the center the arm ready to drop the item on top of the delivery robot's white plate.

5 Discussion and Related Work

Over the last 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). In this section, we review 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.

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 [24], V^3CMM [3], BRICS [9], RoboChart [48], and SafeRobots [53]. Some of them can be used to build ROS-based systems by either supporting a direct translation, e.g., Hyperflex [8], or serving as a base for other platforms. For example, in BRIDE [10], 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 component-based systems are provided in [43]. UML has also been used to model

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 [28]		✓						✓		✓
ATLAS [39]		✓	✓		✓			✓		✓
BRIDE [10]		✓						✓	✓	✓
CommonLang [54]		✓						✓		✓
Drona [23]	✓	✓	✓		✓	✓		✓		✓
FLYAQ [12]		✓	✓							✓
Hyperflex [8]		✓						✓	✓	✓
ISPL [44]	✓	✓	✓		✓	✓		✓		
Koord [34]	✓	✓	✓	✓	✓	✓		✓		✓
PROMISE [32]		✓	✓							✓
RobotChart [48]	✓	✓						✓	✓	
ROSBuzz [55]	✓	✓	✓	✓	✓	✓		✓	✓	✓
RSSM [30]	✓							✓		✓
SCEL [21]	✓	✓	✓	✓	✓	✓	✓	✓		
X-Klaim	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

and design robotic tasks and missions, e.g., Art2ool [28] supports the development cycle of robotic arm tasks in which atomic tasks are abstracted with UML class diagrams. Textual languages, e.g., CommonLang [54], are another type of language used to model robotic systems. For example, in [2], a DSL based on the Python language is presented that can be used interactively, through the Python command-line interface, to create brand new ROS nodes and to reshape existing ROS nodes by wrapping their communication interfaces.

Some other contributions, to some extent, allow for the verification of ROS-based systems. ROSGen [47] takes a specification of a ROS system architecture as an input and generates a ROS node as an output. Using the theorem prover Coq, the generation process is amenable to formal verification. DeROS [1] permits describing a robot's 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 [41], which allows generating C++ code for a monitoring node from user-defined properties specified in terms of event sequences. In [56], robot systems are modeled as a network of timed automata that, after verification in Uppaal, are automatically translated into executable C++ code satisfying the same temporal logic properties as the model. Finally, RSSM [30] permits to model the activities of multi-agent robot systems using Hierarchical Petri Nets and, once deadlock absence has been checked on this model, to generate C++ code for ROS packages automatically.

The approaches mentioned above have not been applied to such complex systems as MRSs, and some of them are not even suitable for such systems. Very few high-level languages for MRSs have been proposed. For example, FLYAQ [12] is a set of DSLs based on UML to specify the civilian missions for unmanned aerial vehicles. This work is extended in [26] for enabling the use of a declarative specification style, but it only supports homogeneous robots. ATLAS [39], which also provides a simulator-based analysis, takes a step further towards coordination of MRSs, but it only supports centralized coordination. PROMISE [32] allows specifying the missions of MRSs using Linear Temporal Logic operators for composing robotic mission patterns. Finally, RMoM [40] 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.

Languages for Coordination. Coordination for MRSs has been investigated from several diverse perspectives, and nowadays there is a wide range of techniques that can be used to orchestrate the actions and movements of robots operating in the same environment [25,57]. 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 [29]. 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 provid-

ing communication and coordination primitives suitable for designing robust MRSs are reviewed in [14]. In ISPL [44], communication is obtained as an indirect result of synchronization of multiple labeled transition systems on a specific action. In SCEL [21], 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 [51], 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 [55]. Differently from X-KLAIM, however, Rosbuzz does not provide high-level coordination primitives, robots' distribution is not explicit, and permits less heterogeneity. Drona [23] is a framework for distributed drones where communication is somehow similar to the one used in ISPL. Koord [34] is a language for programming and verifying distributed robotic applications where communication occurs through a distributed shared memory. Differently from X-KLAIM, however, robots distribution is not explicit, and open-endedness is not supported. Finally, in [46] 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 both 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.

6 Concluding Remarks and Future Work

In this paper, we have presented an approach for programming robotics applications based on the language X-KLAIM and the ROS framework. We have extended the approach introduced in [5] from single robot scenarios to MRS ones. 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.

We believe that 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 robots' 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 both in 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. Moreover, the form of communication offered by tuple spaces, supported by X-KLAIM, benefits the scalability of MRSs in terms of the number of components and robots that can be dynamically added. This would also permit to meet the open-endedness require-

ment (i.e., robots can dynamically enter or leave the system), which is crucial in MRSs.

Our long-term goal is to design a domain-specific language 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 [45]. The tools already developed for KLAIM, e.g., type systems [16,17,37,38], behavioral equivalences [18], flow logic [22], and model checking [19,20,27], 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 [19] has been presented in [36].

Runtime adaptation is another important capability of MRSs. In [35], 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 to check the presence of a matching tuple without being blocked indefinitely. Under 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 these operations return false, and the programmer can adopt 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 [6], 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 noticing 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). Nevertheless, we plan to investigate the definition of a programming framework to make transparent the use of the publish/subscribe mechanism as mentioned above, overcoming the performance issue by elevating the level of abstraction. The idea is not only to replace topics with tuples, but to provide ready-to-use processes acting as building blocks for creating robotics applications. The API for interacting with these processes will be tuples with given structures. These processes will hide the interactions with the ROS framework to the programmer, and produce tuples only when events relevant to the coordination of the MRS behavior occur (e.g., a robot reached a given position or a requested movement has been completed).

Finally, in this work we have used the 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 the version 2 of ROS, which features a more sophisticated publish/subscribe system based on the OMG DDS standard.

Acknowledgements. We thank the anonymous reviewers for their useful comments.

References

1. Adam, S., Larsen, M., Jensen, K., Schultz, U.P.: Rule-based dynamic safety monitoring for mobile robots. J. Softw. Eng. Rob. **7**(1), 121–141 (2016)
2. Adam, S., Schultz, U.P.: Towards interactive, incremental programming of ROS nodes. In: Workshop on Domain-Specific Languages and models for Robotic systems (2014)
3. Alonso, D., et al.: V^3CMM: a 3-view component meta-model for model-driven robotic software development. J. Softw. Eng. Rob. **1**, 3–17 (2010)
4. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java package for distributed and mobile applications. Softw. Pract. Exp. **32**(14), 1365–1394 (2002)
5. Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Writing robotics applications with X-Klaim. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12477, pp. 361–379. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_22
6. Bettini, L., Loreti, M., Pugliese, R.: An infrastructure language for open nets. In: SAC, pp. 373–377. ACM (2002)
7. Bettini, L., Merelli, E., Tiezzi, F.: X-Klaim is back. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 115–135. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21485-2_8
8. Brugali, D., Gherardi, L.: HyperFlex: a model driven toolchain for designing and configuring software control systems for autonomous robots. In: Koubaa, A. (ed.) Robot Operating System (ROS). SCI, vol. 625, pp. 509–534. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26054-9_20
9. Bruyninckx, H., et al.: The BRICS component model: a model-based development paradigm for complex robotics software systems. In: SAC, pp. 1758–1764. ACM (2013)

10. Bubeck, A., et al.: BRIDE - a toolchain for framework-independent development of industrial service robot applications. In: ISR, pp. 137–142. VDE (2014)
11. Casalaro, G.L.: Model-driven engineering for mobile robotic systems: a systematic mapping study. Softw. Syst. Model. (2021). https://doi.org/10.1007/s10270-021-00908-8
12. Ciccozzi, F., et al.: Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems. IEEE Access **4**, 6451–6466 (2016)
13. 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)
14. De Nicola, R., Di Stefano, L., Inverso, O.: Toward formal models and languages for verifiable multi-robot systems. Front. Rob. AI **5**, 94 (2018)
15. 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)
16. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. Theor. Comput. Sci. **240**(1), 215–254 (2000)
17. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. Sci. Comput. Program. **63**(1), 57–87 (2006)
18. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. Inf. Comput. **205**(10), 1491–1525 (2007)
19. De Nicola, R., Katoen, J., Latella, D., Loreti, M., Massink, M.: Model checking mobile stochastic logic. Theor. Comput. Sci. **382**(1), 42–70 (2007)
20. De Nicola, R., Loreti, M.: A modal logic for mobile agents. ACM Trans. Comput. Log. **5**(1), 79–128 (2004)
21. 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), 1–29 (2014)
22. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. **75**(6), 376–397 (2010)
23. Desai, A., Saha, I., Yang, J., Qadeer, S., Seshia, S.A.: Drona: a framework for safe distributed mobile robotics. In: 8th Internnational Conference on Cyber-Physical Systems, pp. 239–248 (2017)
24. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS (LNAI), vol. 7628, pp. 149–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34327-8_16
25. Doriya, R., Mishra, S., Gupta, S.: A brief survey and analysis of multi-robot communication and coordination. In: International Conference on Computing, Communication, Automation, pp. 1014–1021 (2015)
26. Dragule, S., Meyers, B., Pelliccione, P.: A generated property specification language for resilient multirobot missions. In: Romanovsky, A., Troubitsyna, E.A. (eds.) SERENE 2017. LNCS, vol. 10479, pp. 45–61. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65948-0_4
27. 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)
28. 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)
29. 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)

30. Figat, M., Zieliński, C.: Robotic system specification methodology based on hierarchical Petri nets. IEEE Access **8**, 71617–71627 (2020)
31. Frigerio, M., Buchli, J., Caldwell, D.G.: A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In: Proceedings of DSLRob 2011. CoRR, vol. abs/1301.7190 (2013)
32. García, S., et al.: High-level mission specification for multiple robots. In: 12th ACM SIGPLAN International Conference on Software Language Engineering, p. 127–140 (2019)
33. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
34. Ghosh, R., et al.: Koord: a language for programming and verifying distributed robotics application. Proc. ACM Program. Lang. **4**(OOPSLA), 1–30 (2020)
35. Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F.: Modeling adaptation with a tuple-based coordination language. In: SAC 2012, pp. 1522–1527. ACM (2012)
36. Gjondrekaj, E., et al.: Towards a formal verification methodology for collective robotic systems. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 54–70. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_7
37. Gorla, D., Pugliese, R.: Enforcing security policies via types. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) Security in Pervasive Computing. LNCS, vol. 2802, pp. 86–100. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39881-3_10
38. Gorla, D., Pugliese, R.: Resource access and mobility control with dynamic privileges acquisition. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 119–132. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_11
39. Harbin, J., et al.: Model-driven simulation-based analysis for multi-robot systems. In: 24th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2021)
40. Hu, C., Dong, W., Yang, Y., Shi, H., Zhou, G.: Runtime verification on hierarchical properties of ROS-based robot swarms. IEEE Trans. Reliabil. **69**(2), 674–689 (2019)
41. Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., Rosu, G.: ROSRV: runtime verification for robots. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 247–254. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_20
42. Koenig, N.P., Howard, A.: Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: IROS, pp. 2149–2154. IEEE (2004)
43. Kumar, P., et al.: Rosmod: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. In: International Symposium on Rapid System Prototyping (RSP) (2015)
44. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas: an open-source model checker for the verification of multi-agent systems. Softw. Tools Technol. Transfer **19**(1), 9–30 (2017)
45. 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)
46. Majumdar, R., Yoshida, N., Zufferey, D.: Multiparty motion coordination: from choreographies to robotics programs. Proc. ACM Program. Lang. **4**(OOPSLA), 134:1–134:30 (2020)

47. Meng, W., Park, J., Sokolsky, O., Weirich, S., Lee, I.: Verified ROS-based deployment of platform-independent control systems. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 248–262. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_18
48. Miyazawa, A., et al.: RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. **18**(5), 3097–3149 (2019)
49. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. Softw. Eng. Rob. **7**, 75–99 (2016)
50. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 195–206. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_17
51. Pinciroli, C., Lee-Brown, A., Beltrame, G.: A tuple space for data sharing in robot swarms. EAI Endorsed Trans. Collab. Comput. **2**(9), e2 (2016)
52. Quigley, M., et al.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
53. Ramaswamy, A., Monsuez, B., Tapus, A.: SafeRobots: a model-driven approach for designing robotic software architectures. In: Proceedings of CTS, pp. 131–134. IEEE (2014)
54. Rutle, A., Backer, J., Foldøy, K., Bye, R.T.: CommonLang: a DSL for defining robot tasks. In: Proceedings of MODELS18 Workshops. CEUR Workshop Proceedings, vol. 2245, pp. 433–442 (2018)
55. St-Onge, D., Varadharajan, V.S., Li, G., Svogor, I., Beltrame, G.: ROS and Buzz: consensus-based behaviors for heterogeneous teams. CoRR abs/1710.08843 (2017)
56. Wang, R.: A formal model-based design method for robotic systems. IEEE Syst. J. **13**(1), 1096–1107 (2018)
57. Yan, Z., Jouandeau, N., Ali, A.: A survey and analysis of multi-robot coordination. Int. J. Adv. Rob. Syst. **10**, 1 (2013)