




Writing Robotics Applications with X-KLAIM

Lorenzo Bettini¹(✉) , Khalid Bourr²(✉), Rosario Pugliese¹(✉) ,
and Francesco Tiezzi²(✉) 

¹ Dipartimento di Statistica, Informatica, Applicazioni,
Università degli Studi di Firenze, Florence, Italy
{lorenzo.bettini,rosario.pugliese}@unifi.it

² School of Science and Technology, Computer Science Division,
Università di Camerino, Camerino, Italy
{khalid.bourr,francesco.tiezzi}@unicam.it

Abstract. Developing robotics applications is a demanding software engineering challenge. Such a software has to perform multiple cooperating tasks in a well-coordinated manner in order to avoid unsatisfactory behavior. In this paper, we define an approach for developing robot software based on the integration of the programming language X-KLAIM and the popular robotics framework ROS. X-KLAIM is a programming language specifically devised to design distributed applications consisting of software components interacting through multiple distributed tuple spaces. Advantages of using X-KLAIM in the robotics domain derive from its high abstraction level, that allows developers to focus on robots' behavior, and from its computation and communication model, which is especially suitable for dealing with the distributed nature of robots' architecture. We show the feasibility and the effectiveness of the proposed approach by implementing a scenario involving a robot looking for potential victims in a disaster area.

Keywords: Robotics applications · X-KLAIM · Tuple spaces · ROS

1 Introduction

Autonomous robots are versatile machines increasingly used in many fields in today's society, while their capabilities are becoming ever more complex and heterogeneous. They are software-intensive systems, whose software components are typically deployed on a distributed and heterogeneous computing infrastructure, possibly with limited resources. Such software components interact in real-time with a highly dynamic and uncertain environment through sensors and actuators.

Developing robotics applications is currently among the most demanding software engineering challenges [12, 14, 18, 23]. Indeed, such a software has to

The work was supported by the PRIN project "SEDUCE" n. 2017TWRCNB.

© Springer Nature Switzerland AG 2020

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12477, pp. 361–379, 2020.

https://doi.org/10.1007/978-3-030-61470-6_22

perform the multiple cooperating tasks in a well-coordinated manner in order to avoid unsatisfactory behavior that can even cause economic losses and threaten safety. Moreover, since low-level details must be considered in the early phases, robotic experts need very good programming skills or the help of programming experts. In general, expertise from multiple domains needs to be integrated conceptually and technically. Finally, robotic software is difficult to adapt to hardware changes.

In the last few years, a variety of software libraries and middlewares have been specifically developed by different research laboratories and universities to assist and simplify the rapid prototyping of robotic applications. They offer mechanisms for, e.g., real-time control, synchronous and asynchronous communication, abstract access to sensors and actuators. Many researchers have also proposed using higher-level abstractions to drive the software development process and then resorting to some tools for automatic generation of executable code and system configuration files. This permits hiding the lower-level programming details to robotic experts and helping them to focus on their own field of expertise rather than on implementation. The use of a suitably abstract level also supports better maintainability and reusability of software components, and reduces the effort in understanding and modifying the software. Many proposals in the literature are surveyed in [23]. We mention the domain-specific language RobotML [14], enabling to describe robotics concerns with concepts and notations closer to the respective problem domain and to automate code generation. We also mention the prototype framework CommonLang [26], exploiting model-driven software engineering techniques to abstract away from underlying technologies and create executable code for different robotics platforms using code generation.

Along this direction, in this paper we propose an approach for developing robotics applications based on the integration of the programming language X-KLAIM, and its effective Eclipse-based IDE, with the ROS middleware.

X-KLAIM¹ (eXtended KLAIM, originally introduced in [6] and reimplemented from scratch in [8]) is based on the coordination language KLAIM [13] specifically devised to design distributed applications consisting of (possibly mobile) software components interacting through multiple distributed tuple spaces. X-KLAIM code is compiled into Java code and executed on a standard JVM. Because of its specific features, we envisage possible exploitation of the renewed X-KLAIM as a coordination language for developing modern ICT systems, in such domains as robotics, IoT, Smart Cities, e-Health, etc. As a language, X-KLAIM provides a high level of abstraction, allowing developers to focus on robots' behavior while abstracting from technical details (e.g., the low-level commands sent to robots' actuators and the management of events and data coming from robots' sensors). Moreover, as argued below, X-KLAIM features many advantages for different kinds of software architectures used in robotics systems [20]. Its computation and communication model, inherited from KLAIM, is particularly suitable for dealing with the distributed nature of robots architecture, where the components

¹ <https://github.com/LorenzoBettini/xklaim>.

(e.g. actuators and sensors) execute concurrently. Indeed, the X-KLAIM computation model permits to distribute an application across multiple threads of execution or even multiple hardware platforms. Each application component may have its own *tuple space*, that is a repository for storing *anonymous* data and *associatively* retrieving them by means of a pattern-matching mechanism. Application components communicate by means of their distributed tuple spaces, where all data are stored and accessed by the components that are responsible for performing specific tasks. This model features ease of implementation and low computational overhead. It ensures that components can operate *independently*, and gather *asynchronously* the required data by accessing it from a tuple space, without having to communicate directly with each other. If necessary, the same data can be read by multiple components, without the need to replicate them. Appropriate synchronizations among the application components can be implemented still through the tuple spaces. By exchanging request and response messages through the tuple spaces, a component can also act as a service that replies with a response message once another component sends a request message.

ROS² (Robot Operating System [24]) is a well-known set of software libraries and tools to build robotics applications. Since X-KLAIM code is compiled into Java and can interact with any existing Java library, we make use of *java_rosbridge*³ to connect the code generated from an X-KLAIM program with the ROS server that enacts the publish/subscribe interactions of ROS components. This allows us to use X-KLAIM only for writing the code that controls the robot's behavior in a compact and readable way. We also abstract the typical robot behaviors, described at ROS level by large pieces of 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 robotics applications. For illustrating the proposed approach, we consider a simple disaster scenario. To show the execution of the generated code we use Gazebo⁴, 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.

The rest of the paper is organized as follows. In Sect. 2, we provide some background notions concerning the languages and the technologies at the basis of our approach, 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 discuss more strictly related work, while in Sect. 6 we conclude and touch upon directions for future work.

² <https://www.ros.org/>.

³ https://github.com/h2r/java_rosbridge.

⁴ <http://gazebosim.org/>.

2 Background Notions

In this section, we briefly summarize some background notions concerning the languages and the technologies at the basis of our approach. We refer the interested reader to the referred sources for a full account of each of them.

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. Although KLAIM is based on process algebras [22], it builds on the notion of *generative communication* introduced by the coordination language Linda [19] and generalizes it 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 by means of a *pattern-matching* mechanism. Interprocess communication occurs through *asynchronous* exchange of tuples via tuple spaces: processes can indeed *insert*, *read* and *withdraw* tuples into/from tuple spaces. Communicating processes are thus decoupled both in space and time as there is no need for producers (i.e., senders) and consumers (i.e., receivers) of a tuple to synchronize. Tuple spaces are identified by means of *localities*, that are symbolic addresses of network nodes where processes and tuples can be allocated. Localities can be exchanged through interprocess communication. They provide the naming mechanism to represent the notion of administrative domain: computations at a given locality are under the control of a specific authority.

A computational *node* of a KLAIM network is characterized by its locality and a collection of running processes.⁵ *Processes*, i.e., the active computational units of KLAIM, can be executed concurrently, either at the same locality or at different localities. They are built up by composing basic actions acting on network nodes, process variables and process calls, either sequentially or in parallel. Process variables support *higher-order communication*, namely the capability to exchange (the code of) a process 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 pro-

⁵ For the sake of presentation, we omit from the description of KLAIM nodes the distinction between physical and logical localities and, hence, the so called *allocation environment*. The latter is a component of a node that acts as a name solver binding logical localities, occurring in the processes hosted in the node, to specific physical localities.

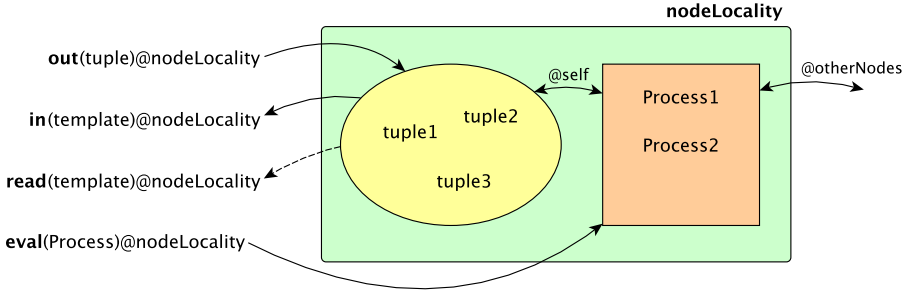


Fig. 1. A KLAIM node.

cesses. In general, any of these fields can contain variables. Instead, an evaluated tuple must not contain variables. Thus, tuple evaluation only succeeds when a tuple does not contain variables and amounts to computing the values of the expressions occurring in the tuple. 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 non-deterministically chosen, otherwise, the process is blocked. These retrieval actions exploit templates as patterns to select tuples in a tuple space. *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 formal fields are left unchanged by the evaluation. Intuitively, an evaluated template matches against an evaluated 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. A successful matching returns a substitution function mapping the variables contained in the formal fields of the template to the values contained in the corresponding actual fields of the accessed tuple. Such a substitution is then applied to the process syntactically following the action. Action `eval(Process)@nodeLocality` sends `Process` for execution to the (possibly remote) node identified by `nodeLocality`. Finally, KLAIM also provides an action for creating new network nodes, but we do not present it here as it is not exploited in the paper.

2.2 KLAVA and X-KLAIM

The implementation of KLAIM basically consists of two main components:

- the Java package KLAVA (KLAIM in Java, originally introduced in [4]);
- the programming language X-KLAIM.

KLAVA provides the implementation of the KLAIM tuple space operations and concepts (such as *nodes*, *nets*, *processes*, etc.) in terms of classes and methods, relying on the IMC framework [3] 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 allows Java programmers to fully exploit Java mechanisms and the libraries of its huge ecosystem, while using the KLAIM programming model. However, programmers have to deal with the verbosity of Java, which also makes it hard to directly use KLAIM primitives. KLAVA strives for making Java programmers' life easier, but it has to obey the rules of Java. For this reason, we also developed X-KLAIM, a domain-specific language that is closer to KLAIM while providing typical high-level programming constructs. The X-KLAIM compiler translates X-KLAIM programs into Java code that uses the Java package KLAVA. The produced Java code can be then compiled and executed using the standard Java toolchain.

The versions of X-KLAIM and KLAVA used in this paper are available as an open source project. Sources and links to Eclipse update site and to complete Eclipse distributions are available from: <https://github.com/LorenzoBettini/xklaim>.

For the new implementation of X-KLAIM we relied on XTEXT [5], an Eclipse framework for the development of programming languages and DSLs. XTEXT also provides a complete IDE support based on Eclipse: editor with syntax highlighting, code completion, error reporting and incremental building, just to mention a few. Furthermore, we made use of another mechanism provided by XTEXT, that is, XBASE [17], an extensible and reusable expression language. By using XBASE in X-KLAIM, besides a rich Java-like syntax, we also inherit its interoperability with Java and its type system. In fact, an X-KLAIM program can seamlessly access any Java type and Java library available in the classpath of the project. The interoperability with Java allowed us to seamlessly integrate X-KLAIM with *java_rosbridge*.

The syntax of XBASE is similar to Java, thus it should be easily understood by Java programmers, but it removes much “syntactic noise” from Java. For example, terminating semicolons are optional, as well as other syntax elements like parenthesis when invoking a method without arguments. Moreover, XBASE comes with a powerful type inference mechanism, compliant with the Java type system: the programmer can avoid specifying types in declarations when they can be inferred from the context. 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 turns triggers the generation of Java byte-code. Notably, the X-KLAIM integration in Eclipse, allows the programmer to debug an X-KLAIM program.

In the rest of this section we briefly describe the main features of X-KLAIM that are relevant for this paper. Thus, for example, we will not describe code mobility features, as they are not used in this paper (the interested reader is referred to [8] for more details).

An X-KLAIM program can contain process, node and net definitions. All these components can also be defined in separate files and can be referred through a Java-like *import* mechanism.⁶

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 KLAIM operations (that we will show in Sect. 4). Typical programming structures such as *if*, *while* and OOP Java-like mechanisms, such as object creation and method invocation, are already part of XBASE.

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

```
net ANet {
  node Node1 { ... start code ... }
  node Node2 { ... start 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 we will see in the example 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 [7], but we will not use it in this paper.

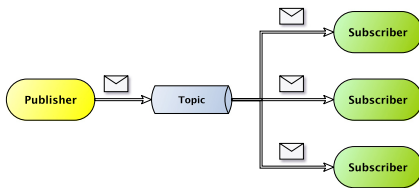


Fig. 2. ROS publish/subscribe mechanism.

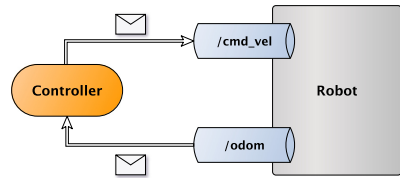


Fig. 3. Interaction with ROS robot.

2.3 ROS

Robotic Operating System (ROS)⁷ is one of the most sophisticated and popular frameworks for writing robot software. It provides tools and libraries for simplifying the development of complex and robust robot controllers while abstracting from the underlying hardware. ROS works with more than a hundred robots,

⁶ Code completion is provided in the X-KLAIM Eclipse editor for imports as well as standard “Organize imports” mechanisms.

⁷ <https://www.ros.org/>.

ranging from autonomous cars to drones and humanoid robots, and integrates a multitude of 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 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. 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. This decouples the production of data from its consumption. Multiple publishers and subscribers for the same topic are allowed. The diagram in Fig. 2 illustrates this concept, while the one in Fig. 3 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: it sends a message directed to the wheels actuator and receives back a message containing the position the robot has moved to. The topic `/cmd_vel` stands for *command velocity*. The topic `/odom` stands for *odometry*, the technique used to estimate the change in position over time from robot sensors data.

3 Our Approach and Framework

In this section we illustrate our approach, and the resulting software framework, for programming robotics applications using X-KLAIM and ROS.

The architecture of autonomous robots has a distributed nature, as it typically consists of different components, in particular sensors and actuators, that cooperate with each other making use of a communication infrastructure. Their software architecture reflects such a distribution and partitions the robot's software into parts, with specific relationships among them, working together as a coherent whole. Robot components are thus managed by specialized processes that may need to work on local data and can demand dedicated machines for their execution.

This distributed architecture of the robot's software is naturally rendered in X-KLAIM as a network where the different parts are deployed. As depicted in Fig. 4, we typically have a controller node and several sensor and actuator nodes. The latter nodes are not fixed once and for all. Rather, they can be dynamically added or removed, and even equipped with different processes, in order to represent different robot types and configurations. To concretely program with the X-KLAIM language the behaviors of robots, we have integrated it with the ROS middleware. The communication infrastructure of the integrated framework is graphically depicted in Fig. 5.

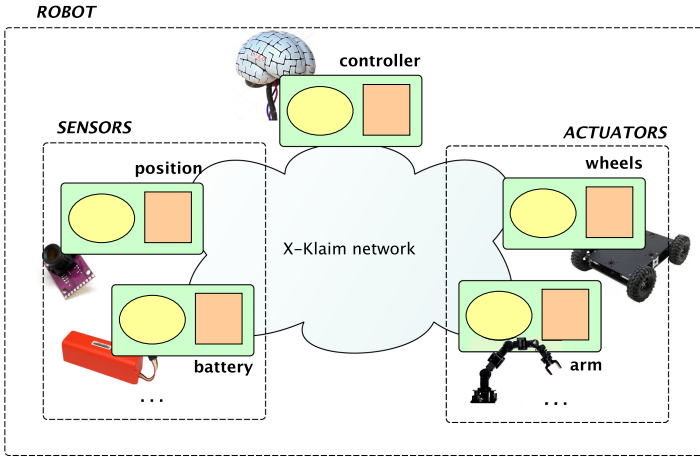


Fig. 4. Software architecture of robots in X-KLAIM.

Specifically, X-KLAIM applications are indirectly connected with the ROS framework by means of the Java library *java_rosbridge*. It provides Java objects supporting publishing and subscribing over ROS topics. In its own turn, *java_rosbridge* communicates with the ROS Bridge server, via the WebSocket protocol, by means of the Jetty web server.⁸ The ROS Bridge server, indeed, provides via WebSocket a JSON API to ROS functionality for external programs. This way, ROS receives and executes commands on the physical robot, and gives feedback and sensor data. In addition, ROS can optionally interact with the Gazebo simulator,⁹ via the ROS commutation mechanism (e.g., by launching the simulator as a ROS node). The use of the simulator is not mandatory when ROS is deployed in a real robot; however, even in such a case, the

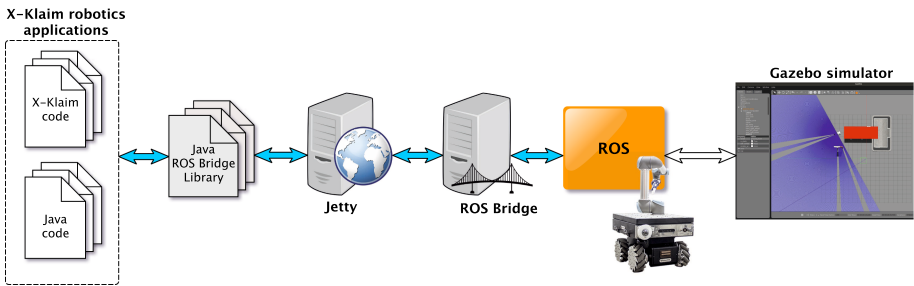


Fig. 5. The integrated framework.

⁸ Jetty 9: <https://www.eclipse.org/jetty/>.

⁹ This interaction is denoted in Fig. 5 by a white arrow, to stress its optionality.

```

{ "topic": "/robot_base_velocity_controller/cmd_vel",
  "msg": {
    "linear": { "x": 0.0013, "y": 0.0, "z": 0.0 },
    "angular": { "x": 0.0, "y": 0.0, "z": 0.0029 }
  }
}

```

Fig. 6. Example of a JSON message for the `/cmd_vel` topic.

design activity of the robot’s controller may benefit from the use of a simulator, to save time and reduce the development cost.

A crucial role in the framework described above is played by JSON messages. Indeed, the use of JSON enables the interoperability of ROS with most programming languages, including Java. As an example, we report in Fig. 6 a Twist message in the JSON format published on the ROS topic `/cmd_vel`, providing information for moving the robot. This message expresses the velocity in terms of its linear and angular parts, each of which defined as a vector.

4 X-KLAIM at Work on a Robotics Scenario

For illustrating the proposed approach, in this section we show and briefly comment a few interesting parts of the implementation of a simple robotics scenario. The full source code can be found at <https://github.com/LorenzoBettini/xklaim-ros-example>. It consists of an Eclipse/Maven project with X-KLAIM code (and its generated Java code), using *java_rosbridge*.¹⁰

The scenario that we consider involves a robot looking for potential victims in a disaster area. By following a random walk, the robot explores an unknown, flat environment where a number of obstacles are present while avoiding collisions with them. As soon as the robot has localized a potential victim, it stops near the victim and signals its position. The robot has a limited battery lifetime and the battery’s state of charge is monitored during the course of the robot’s activities. If the state of charge drops under a given threshold value, then the robot stops searching for a victim and rather moves towards a charging station whose position is known to it.

In Fig. 7 we show the whole network for our implementation of the scenario. As discussed in Section 3, each part of the robot is rendered as an X-KLAIM node, whose name represents its locality (see Sect. 2.2). For each node we have one or several processes that deal with the robot’s sensors (e.g., `PositionSensor`) or with the robot’s moving parts (e.g., `WheelsActuator`). Each node creates processes locally and executes them concurrently by means of the KLAIM operation `eval`. We have made a few processes parametric with respect to the localities, so that they can be easily relocated to any node. This way, we could experiment with different network configurations (see also Sect. 6). The most interesting

¹⁰ We ‘consume’ *java_rosbridge* and X-KLAIM runtime libraries as Maven artifacts.

```

net RobotNet {
  node Position {
    eval(new PositionSensor())@self
  }
  node Wheels {
    eval(new WheelsActuator())@self
    eval(new GoToActuator(Position))@self
  }
  node Controller {
    eval(new RobotController(Wheels, Position, Battery, ObstacleAvoidance))@self
  }
  node Battery {
    eval(new LowBatterySensor(Controller))@self
    eval(new BatteryConsumption())@self
    eval(new BatteryCharge(Position))@self
  }
  node ObstacleAvoidance {
    eval(new LaserSensor())@self
    eval(new CalculateOrientation(Wheels))@self
  }
  node Victim {
    eval(new VictimSensor(Controller, Position))@self
  }
}

```

Fig. 7. The X-KLAIM net of our example.

process under that respect is `RobotController`, which has to deal with several robot's components and so it takes such components' localities as parameters.

The source code of the process `RobotController` is shown in Fig. 8. The code should be easily readable by a Java programmer. Such types as `Double`, `Locality` and `Random` (note the `import` statement) are actually Java types, since, as mentioned above, X-KLAIM programs can refer directly to Java types. Note also that `nextIntFloat` is actually the Java method of the Java class `Random`. Java static methods, like `String.format`, can be used as well; `println` is a shortcut for the standard `System.out.println`. Variable declarations in XBASE 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. Note how XBASE removes much syntactic noise of Java. It also treats safely operators such as `==`, which, for objects like `String`, actually translates to a call `equals` in the generated Java code. 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

```

import java.util.Random

proc RobotController(Locality wheels, Locality position,
    Locality battery, Locality obstacleAvoidance) {
    val rand = new Random()
    out("control step", "random walking")@self // initial control step tuple
    while (true) {
        read("control step", var String stepType)@self // read the control step tuple
        if (stepType == "random walking") {
            val angularVelocity = rand.nextFloat() * 10 - 5 // a random angular velocity
            val linearVelocity = 1.5
            out("velocity", linearVelocity, angularVelocity)@obstacleAvoidance
        } else if (stepType == "low battery") {
            read("charge station position", var Double x, var Double y)@battery
            out("go to", x, y)@wheels
            in("battery charged")@battery
            in("control step", stepType)@self
            out("control step", "random walking")@self
        } else if (stepType == "victim detected") {
            out("velocity", 0.0, 0.0)@wheels
            read("position", var Double x, var Double y, var Double anyTheta)@position
            println(String.format("victim detected at position: %f, %f", x, y))
            return
        }
    }
}

```

Fig. 8. The X-KLAIM RobotController process.

declare variables that are available in the code after **in** and **read** operations (just like in KLAIM).¹¹

Besides that, the code of Fig. 8 basically relies on the KLAIM tuple space based communication. The controller first reads a local tuple containing the “type” of step to perform and acts accordingly. The ‘normal’ behavior consists of random walking in the working area. The controller creates the tuple indicating the velocity broken in its linear and angular part, and inserts it in the obstacle avoidance’s tuple space. If the level of the robot’s battery is too low, the robot goes to a charging station. The controller retrieves the charge station position, moves the robot to the charge station and waits for the completion of the charge. Then, it replaces the low battery control step with the random walking one. If a victim is found, the controller stops the movement by sending velocity 0 to the wheels actuator. It then sends the current position to the rescuers (here it simply prints out a message in the console with the position of the victim) and the process terminates. During these actions the controller communicates with other

¹¹ Non-blocking versions of **in** and **read** are also available: **in_nb** and **read_nb**, respectively.

```

import ros.*
import ros.msgs.geometry_msgs.Twist

proc WheelsActuator() {
  val bridge = new RosBridge()
  bridge.connect("ws://0.0.0.0:9090", true)
  val publisher = new Publisher(
    "/robot_base_velocity_controller/cmd_vel", "geometry_msgs/Twist", bridge)
  while (true) {
    in("velocity", var Double x, var Double z)@self
    val twistMsg = new Twist();
    twistMsg.linear.x = x
    twistMsg.angular.z = z
    publisher.publish(twistMsg);
  }
}

```

Fig. 9. The X-KLAIM `WheelsActuator` process.

parts of the robot (i.e., with the corresponding X-KLAIM processes) by means of tuples inserted into or retrieved from specific tuple spaces, whose localities are received as parameters. The localities `wheels` and `positions` correspond to the next two processes we are going to describe.

The code of the process `WheelsActuator` is shown in Fig. 9. Here we can see that X-KLAIM code can also interact with Java libraries, like `java_rosbridge`. In fact, we establish a bridge with the ROS Bridge WebSocket. In this case, we create a ROS publisher (see Sect. 2.3) and we publish `Twist` messages (as the one in Fig. 6). We do that after consuming a tuple containing the velocity data.

The code of the process `PositionSensor` is shown in Figure 10.

As before, we use the Java API provided by `java_rosbridge`. This time we subscribe for a specific topic (we refer to `java_rosbridge` documentation for the used API). The last argument is an XBASE lambda. XBASE *lambda expressions* have the shape: `[param1, param2, ... | body]`. 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 “position” and “orientation”, performs some computation (again, by using the standard Java library) and uses the computed information to update the tuple space. The JSON message format is dictated by ROS. On the contrary, for the tuples inserted in the tuple space, we could have also defined a Java class, e.g., `RobotPosition`, as a datatype for “position” tuples. Indeed, as explained in Section 2.2, any Java object can be inserted in a tuple.

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 the port 9090. Similarly,

```

proc PositionSensor() {
  val bridge = new RosBridge()
  bridge.connect("ws://0.0.0.0:9090", true)
  out("position", 5.0, 4.5, 0.0)@self // initial position and orientation tuple
  bridge.subscribe( // Subscribe to the Pose topic of the robot
    SubscriptionRequestMsg.generate("/robot_base_velocity_controller/odom")
      .setType("nav_msgs/Odometry").setThrottleRate(1).setQueueLength(1),
    [ data, stringRep ]
  )
  val pose = data.get("msg").get("pose").get("pose")
  val position = pose.get("position")
  val x = position.get("x").asDouble()
  val y = position.get("y").asDouble()
  val orientation = pose.get("orientation")
  val qx = orientation.get("x").asDouble()
  val qy = orientation.get("y").asDouble()
  val qz = orientation.get("z").asDouble()
  val qw = orientation.get("w").asDouble()
  val siny_cosp = 2 * (qw * qz + qx * qy)
  val cosy_cosp = 1 - 2 * (qy * qy + qz * qz)
  val theta = Math.atan2(siny_cosp, cosy_cosp)
  in("position", var Double anyX, var Double anyY, var Double anyTheta)@self
  out("position", x, y, theta)@self
}
}

```

Fig. 10. The X-KLAIM PositionSensor process (imports are omitted).

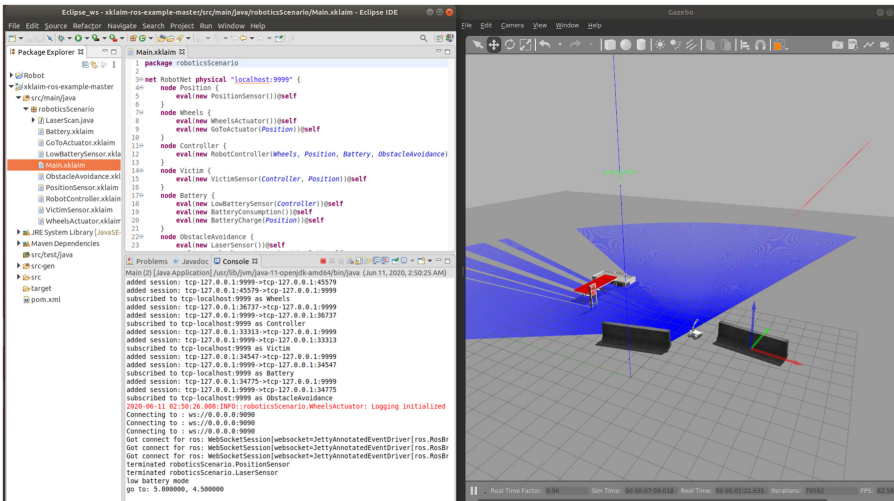


Fig. 11. Execution of an X-KLAIM robotics application.

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 has been generated by the X-KLAIM compiler. The screenshot in Fig. 11 shows our X-KLAIM robotics application in execution. Of course, since the robot explores the disaster area randomly, executions are different from each other.

5 Related Work

More strictly related works are a couple of proposals using high-level languages for producing ROS applications. In [1], an approach is proposed that aims at creating nodes of ROS applications using a DSL based on the Python language. This DSL 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. In [21], the tool ROSGen is described, which, given as an input a specification of a ROS system architecture, generates a ROS node model. This is a glue code written in a DSL, which specifies the ROS nodes that compose the system and the topics that the nodes subscribe to and publish on. This paper also proposes a demonstration that the code generation process is amenable to formal verification, using the theorem prover Coq.

Other less specific works regard applications of Model-Driven Engineering (MDE) and development of DSL for robotics applications. MDE [28] is considered by many robotics researchers to be a promising approach for simplifying design, implementation and execution of software for robotics systems. MDE advocates the use of domain-specific modeling languages (DSMLs) for expressing robotics models through concepts that abstract away from the underlying technology and are closer to the problem domain. Many proposals in the literature instantiate this approach, like e.g. the domain-specific language RobotML [14], the model-based framework SafeRobots [25] and the prototype framework CommonLang [26]. In [15], a family of domain-specific languages for specifying missions of multi-robot systems is introduced. The proposed languages are organized in different layers comprising languages conceived for the end-user describing missions and the environmental context, an intermediate language describing the detailed behavior of each robot (hidden to the user), and the robot language containing the hardware and low-level specification of each type of robot within the team. The authors claim that the layer managing the robot controller is implemented using Java and ROS, which interact via *java_rosbridge* as in our work. In [16], a domain-specific language for developing robot arm applications is defined. Special attention is paid to the automatic generation of robot control logic and to the validation and certification of software components. In [27], the relationships between MDE and the service-oriented component-based development approach in the robotics domain are discussed, and a software engineering approach, called SmartSoft, resulting from their combination is illustrated. All required and provided services of a SmartSoft component are built on top of a

small set of communication patterns, which connect the externally visible services with the internally visible set of access methods for these services. This provides a completely middleware-independent view on the component ports and on the communication interfaces visible to the user. From the modeling perspective, the approach is supported by a UML-based notation, called SmartMARS, representing the SmartSoft concepts independently of any implementation technology. In [2], a model-driven toolchain for robotics software development, based on the 3-View Component Meta-Model V³CMM, is introduced. It provides designers with an expressive, yet simple, platform-independent modeling language for component-based application design. The MDE approach permits generating the code for specific platforms via model transformations, which allow programmers to progressively include application-dependent details. In [10], the BRICS model-based development paradigm is proposed. This paradigm aims at providing robotics developers with a set of guidelines, meta-models and tools for structuring the development of robotics software systems, without introducing any framework or application-specific details. In [9], several MDE-based solutions for software development in robotics are illustrated. This paper focuses specifically on the architectural model as the central artifact of almost all software development activities. In [23], the state of the art in DSMLs in robotics is surveyed. This paper also provides an overview of subdomains relevant for programming and simulation of robotics applications that are already supported through the MDE approach. Finally, [29] shows that by relying on a suitably designed transformation and verification architecture it is possible, also in such critical environments like robotics systems, to mitigate the additional risk resulting from the automatic transformation from DSLs to code through the use of so-called *language workbenches*.

We leave for future work a systematic comparison with the related literature, also aimed at identifying the requirements of robotic applications and showing the benefits of our approach. Anyhow, we want to point out that our work differs from the ones discussed above for the use of a high-level language with a tuple-based communication mechanism. X-KLAIM computation and communication model is particularly suitable for programming robot's behavior. Indeed, X-KLAIM natively supports concurrent programming, which is required by the distributed nature of robots' software. In addition, communicating processes are decoupled both in space and time and 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.

6 Concluding Remarks and Future Work

In this paper we have introduced an approach for developing robotics applications based on the programming language X-KLAIM and the ROS middleware. We consider this as a first exploratory attempt. We think that X-KLAIM has proved expressive enough to implement this first scenario. In particular,

X-KLAIM integration with Java allowed us to seamlessly use the *java_rosbridge* API directly in the X-KLAIM code.

Further experimentation will be needed with application scenarios typical of the robotics domain to allow us to assess whether the linguistic primitives of X-KLAIM are already sufficiently expressive or whether we need to equip the language with further abstractions which better fit the problem domain. In addition, a usability evaluation of the X-KLAIM language will be also needed to assess the benefits of using it in place of the traditional solutions for ROS-based robotics applications (i.e., C++ and Python). Our long-term goal is the design of a domain specific language for the robotics domain which, besides being used for generating executable code, is integrated with automated reasoning tools that can support application verification and analysis.

We also plan to extend our approach from single robot scenarios to collective ones. In this respect, we believe that the form of communication offered by tuple spaces, supported by X-KLAIM, which permits decoupling communicating processes both in space and time, brings benefits for the scalability of collective robotics systems in terms of the number of components and robots that can be dynamically added. This would also permit to meet the open-endedness requirement (i.e., robots can dynamically enter or leave the system), which is crucial in collective systems. The tuple space-based paradigm supported by X-KLAIM relies on KLAVA, which abstracts from the actual implementation of the tuple space. KLAVA itself provides a default implementation where all tuples are stored in a list, which has to be scanned sequentially when looking for a matching tuple. Other optimized and ad-hoc implementations of tuple spaces can be injected into KLAVA. We plan to experiment with such optimizations along the lines of [11].

In the extension from single to multi robots systems, we can take advantage of the hierarchical version of the network model presented in [7], which is already implemented in X-KLAIM, as mentioned in Sect. 2.2. For example, this feature will allow us to organize the components of a single robot in a flat network (as in the current implementation of the example), and to structure the collective system as a network of networks. Note that, as we stressed in Sect. 4, even in the current shape, our processes are already independent from the actual physical positions of the nodes, since they are parametric with respect to tuple space localities.

Since runtime adaptation is another important capability of collective systems, we also plan to investigate to what extent we can benefit from X-KLAIM code mobility mechanisms 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.

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 extend our approach to the version 2 of ROS, which features a more sophisticated publish/subscribe system based on the OMG DDS standard.

References

1. Adam, S., Schultz, U.P.: Towards interactive, incremental programming of ROS nodes. In: Workshop on Domain-Specific Languages and Models for Robotic Systems (2014)
2. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Álvarez, B.: V³CMM: a 3-view component meta-model for model-driven robotic software development. *J. Softw. Eng. Rob.* **1**, 3–17 (2010)
3. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A flexible and modular framework for implementing infrastructures for global computing. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 181–193. Springer, Heidelberg (2005). https://doi.org/10.1007/11498094_17
4. Bettini, L., De Nicola, R., Pugliese, R.: KLAVA: a Java package for distributed and mobile applications. *Softw. Pract. Experience* **32**(14), 1365–1394 (2002)
5. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
6. Bettini, L., De Nicola, R., Pugliese, R., Ferrari, G.L.: Interactive mobile agents in X-Klaim. In: WETICE, pp. 110–117. IEEE Computer Society (1998)
7. Bettini, L., Loreti, M., Pugliese, R.: An infrastructure language for open nets. In: SAC, pp. 373–377. ACM (2002)
8. 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
9. Brugali, D.: Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics. *IEEE Robot. Autom. Mag.* **22**(3), 155–166 (2015)
10. Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G.K., Gherardi, L., Brugali, D.: The BRICS component model: a model-based development paradigm for complex robotics software systems. In: SAC, pp. 1758–1764. ACM (2013)
11. Buravlev, V., De Nicola, R., Mezzina, C.A.: Evaluating the efficiency of Linda implementations. *Concurr. Comput. Pract. Exp.* **30**(8) (2018)
12. De Nicola, R., Di Stefano, L., Inverso, O.: Toward formal models and languages for verifiable multi-robot systems. *Front. Rob. AI* **5**, 94 (2018)
13. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.* **24**(5), 315–330 (1998)
14. Dhoubib, S., Kehir, 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
15. Di Ruscio, D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. In: Proceedings of MORSE@STAF. CEUR Workshop Proceedings, vol. 1319, pp. 16–29 (2014)
16. Djukic, V., Popovic, A., Tolvanen, J.: Domain-specific modeling for robotics: from language construction to ready-made controllers and end-user applications. In: Proceedings of MORSE@RoboCup, pp. 47–54. ACM (2016)
17. Efftinge, S., et al.: Xbase: implementing domain-specific languages for Java. In: GPCE, pp. 112–121. ACM (2012)

18. 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'11. CoRR, vol. abs/1301.7190 (2013)
19. Gelernter, D.: Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
20. Houliston, T., et al.: NUClear: a loosely coupled software architecture for humanoid robot systems. *Front. Rob. and AI* **3**, 20 (2016)
21. 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
22. Milner, R.: *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
23. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Rob.* **7**, 75–99 (2016)
24. Quigley, M., et al.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
25. 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)
26. Rutle, A., Backer, J., Foldøy, K., Bye, R.T.: CommonLang: a DSL for defining robot tasks. In: Proceedings of MODELS 2018 Workshops. CEUR Workshop Proceedings, vol. 2245, pp. 433–442 (2018)
27. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: *Introduction to Modern Robotics*, pp. 119–150. iConcept Press (2011)
28. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. *IEEE Comput.* **39**(2), 25–31 (2006)
29. Voelter, M.: Using language workbenches and domain-specific languages for safety-critical software development. In: Proceedings of SE/SWM. LNI, vol. P-292, pp. 143–144. GI (2019)