



Translating BPMN models into X-KLAIM programs for developing multi-robot missions

Khalid Bourr¹ · Francesco Tiezzi² · Lorenzo Bettini² · Stefano Seriani¹

Accepted: 13 December 2025 / Published online: 14 January 2026

© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2026

Abstract

The development of software for multi-robot systems remains a complex and demanding task, primarily due to the challenges of coordinating interactions among robots while managing low-level implementation details. These complexities can pose significant barriers for non-expert users and hinder effective collaboration between roboticists, domain experts, and stakeholders. To address these issues, we introduce a model-driven development approach for multi-robot systems that leverages BPMN and X-KLAIM. The BPMN standard offers an intuitive graphical notation widely adopted in both industry and academia, enabling the design of sophisticated robotic interactions without requiring deep programming expertise. Meanwhile, X-KLAIM provides a robust framework for implementing multi-robot behaviors, featuring coordination primitives for both intra- and inter-robot interactions and seamless integration with Java for low-level programming. Our approach combines the clarity of BPMN with the operational power of X-KLAIM through an automatic translation of BPMN models into X-KLAIM code. We present the B2XKlaim tool, which automates this translation process, and demonstrate its effectiveness through a case study in the e-agriculture domain. Additionally, we discuss potential applications in collaborative, multi-agent robotic space exploration.

Keywords Multi-robot systems · BPMN · X-Klaim · Model-driven development

1 Introduction

The advent of Multi-Robot Systems (MRSs) has revolutionized numerous industries, enabling enhanced efficiency and innovation in applications ranging from agricultural management to search-and-rescue missions. MRSs are composed of autonomous robots that interact in real-time with highly dynamic and uncertain environments through sensors and actuators. However, the complexity of designing, programming, and managing the intricate interactions within MRSs remains a substantial challenge. Robust and intuitive design

tools are essential to tame this complexity and make MRSs more accessible and effective [21, 28, 40].

To address these challenges, the Business Process Model and Notation (BPMN) standard [41] has emerged as a promising solution in this domain [12, 15, 17]. BPMN offers a graphical notation for specifying the mission of an MRS as a business process, i.e., a workflow model characterized by structured activities undertaken by one or more parties pursuing a common goal. Its strength lies in providing clear and structured diagrams that are easy for stakeholders to understand, fostering a collaborative environment for robotic mission design. Indeed, BPMN collaborations are highly expressive and encapsulate in a unique diagram the interplay among control-flow, data-flow and communication in a flexible and understandable way, making the modeling activity for MRS missions easier [12]. The resulting models are easy to read by humans and, at the same time, suitable for model-to-code solutions. However, the high level of abstraction of BPMN models makes it difficult to specify the technical details required for execution. Moreover, making BPMN models directly executable typically requires integrating a BPMN engine into the robotic context, which can be impractical or inefficient. These issues limit the utility of

✉ K. Bourr
KHALID.BOURR@dia.units.it

F. Tiezzi
lorenzo.bettini@unifi.it

L. Bettini
francesco.tiezzi@unifi.it

S. Seriani
sseriani@units.it

¹ University of Trieste, Trieste, Italy

² University of Florence, Firenze, Italy

BPMN for MRS programming, especially when bridging the gap between conceptual design and operational deployment.

Conversely, X-KLAIM [7] is a programming language specifically tailored for distributed systems and provides the level of detail required for direct execution in MRSs. Its integration with the Robotic Operating System (ROS) framework [42] demonstrates its potential for MRS programming [8–10]. Nevertheless, the steep learning curve and the need for precise control over process interactions can hinder its widespread adoption, particularly among beginner roboticians, domain experts, and other stakeholders who may not have a strong programming background.

To bridge this gap, we propose a model-driven approach that translates BPMN models into X-KLAIM programs. Our translation captures the core design principles and logic of the BPMN model, generating skeleton X-KLAIM code that can be further refined with low-level details for execution on top of the ROS framework. This approach elevates MRS programming from a purely technical task to an intuitive, design-centric process. By retaining the high-level abstraction of BPMN, our approach enables a collaborative development environment, allowing robotics experts, application domain experts, and other stakeholders to jointly design MRS missions. At the same time, the seamless translation to X-KLAIM leverages its executional capabilities while preserving the original design intent. It is worth noticing that, considering the various degrees of abstraction in developing MRS missions, X-KLAIM is positioned between BPMN and ROS code. Indeed, X-KLAIM provides coordination constructs that simplify the coding of robotic systems, especially in the presence of multiple robots [10]. Thus, using X-KLAIM as the target language of the translation makes the refinement of the generated skeleton code easier for the final user while maintaining reasonable execution performance. Summarizing, our proposal bridges the gap between high-level mission planning and low-level mission execution, achieving a balance between conceptual design and operational detail.

To demonstrate the feasibility of our model-driven approach, we developed B2XKlaim, a prototypical software tool that automates the BPMN-to-X-KLAIM translation and provides a user-friendly interface that shields users from underlying complexities. We illustrate the effectiveness of our approach through a practical case study in the e-agricultural domain.

Contribution This paper extends [13] in several ways.

- We extend the fragment of BPMN considered in our translation approach, specifically including timer events and the event-based gateway.
- We discuss mechanisms to address the maintainability of coexisting generated and manual code.

- We explore a series of possible use cases in the field of space exploration, highlighting the generality of our approach.
- We introduce a running example, used throughout the paper to illustrate the concepts and technicalities of our proposal.
- We provide a more detailed description of the languages and technologies at the basis of our approach.

Structure of the paper The rest of the paper is organized as follows. In Sect. 2, we recall background notions on the languages and technologies used in our approach, while in Sect. 3, we describe our translation process. Section 4 presents the approach at work on an e-agriculture scenario involving an MRS composed of a drone and a tractor. Section 5 describes the B2XKlaim tool. Section 6 illustrates possible space exploration use cases. Finally, Sect. 7 discusses related work, and Sect. 8 concludes and outlines directions for future work.

2 Background notions

In this section, we introduce the languages at the basis of our approach: the fragment of BPMN used to model MRS missions, the programming language X-KLAIM, and the framework for robotics applications, ROS.

2.1 BPMN collaboration diagrams

BPMN [41] is a well-established standard for graphically modeling business processes, which facilitates effective communication among stakeholders involved in the design and execution of these processes. Collaboration diagrams, an essential aspect of BPMN, are used to represent the interactions and behaviors of distinct parties working together to achieve a common goal. These diagrams are particularly useful in modeling systems with multiple interacting components, such as MRSs.

A collaboration diagram comprises one or more pools representing entities participating in the collaboration. Each *pool* may contain a process that defines the potential behaviors an entity can exhibit. A pool marked by the symbol ||| represents multiple entities with the same behavior, enabling scalability in the modeling of large-scale systems. A *process* is a workflow consisting of elements, like activities, events, and gateways, connected via sequence flows. Data objects are associated with activities and events through data associations.

To effectively model the missions of MRSs, a carefully selected subset of BPMN elements (Fig. 1) has been identified based on their relevance and applicability to MRS scenarios [12]. These elements facilitate the modeling of

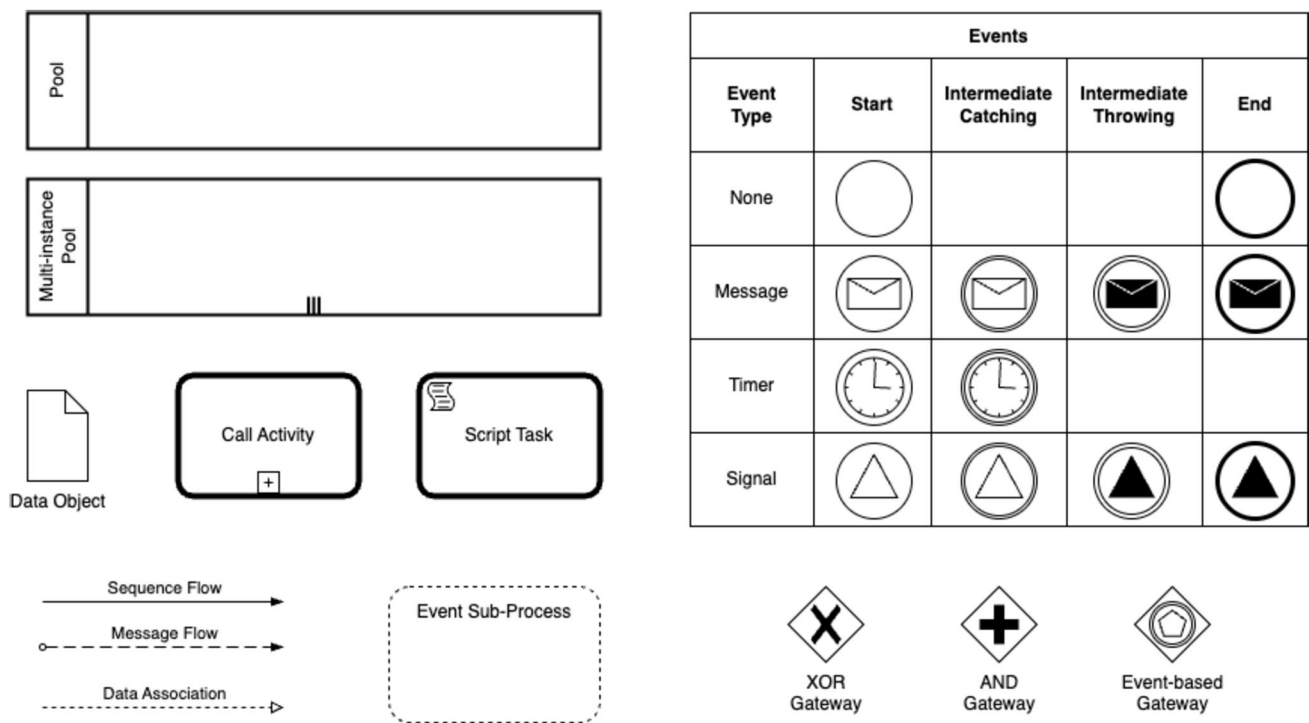


Fig. 1 Selected BPMN elements

interactions and behaviors among multiple robots collaborating to achieve a shared objective. In the context of MRSs, *robots are represented as pools*, with heterogeneous robots depicted as single-instance pools and homogeneous robots as multi-instance pools. This strategic use of multi-instance pools allows for the depiction of identical missions across multiple robots, maintaining clarity and manageability even as the model scales. *The mission of a robot is portrayed as a process* within the pool. The process consists of actions in the form of tasks call activities, and event handlers as event sub-processes. Notably, a call activity references an activity defined in a process that is external to the current process. This allows the creation of multi-layer models, which are particularly useful for dealing with larger models and reusing the same process definition multiple times. Various aspects of a robot mission, such as *internal choices*, *concurrent behaviors*, and *external choices* are represented using *XOR*, *AND*, and *event-based* gateways, respectively. *Communication between robots is modeled via message and signal events*. The former kind of event describes a point-to-point form of communication, while the latter describes a broadcast. Both forms of communication are asynchronous; hence, the sending elements have a non-blocking semantics, while the receiving ones are blocking. Finally, data utilized in mission execution is denoted as data objects.

We conclude this section by illustrating a BPMN collaboration model (in Fig. 2) of a simple robotics scenario aimed at clarifying how the BPMN notation permits modeling MRS

missions. We will use this scenario as a running example throughout the paper.

The scenario involves two robots: an *explorer* robot, whose goal is to explore a territory in search of a given area where to perform a job, and a *worker* robot, whose goal is to help the explorer in performing the work. The explorer mission starts with the Explore call activity. This calls a structured activity, represented as a separate process (omitted here) consisting of a sequence of random movements followed by a soil analysis, repeated until the searched area is found. When the exploration completes, the robot sends (by means of a message intermediate throwing event send position) its current position to the worker, whose mission starts with the reception of this message (by means of the message start event get position). The explorer execution is then blocked (in an event-based gateway), waiting for a message from the worker or the expiration of a timeout (represented by a timer event of 30 seconds). The worker evaluates the received position (by means of the script task Evaluate Position) and decides (via the XOR gateway) whether to send a join notification message or a not join notification one. In the positive case, the worker moves to the explorer's position, notifies its arrival and works. On the other side, the explorer starts either the Work Alone or the Work Together activity depending on the received message (or whether the

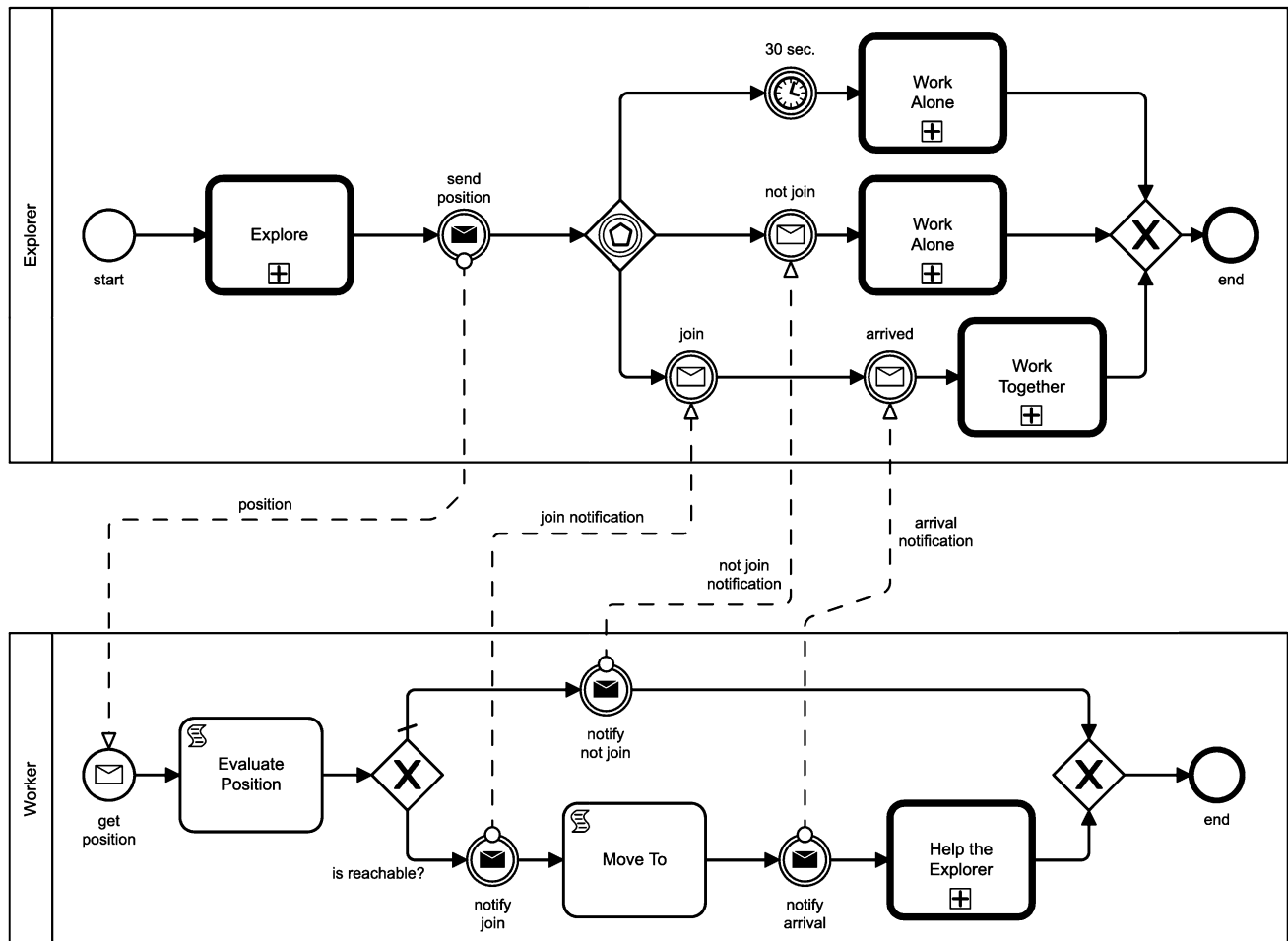


Fig. 2 Running example: BPMN model

timeout has expired).¹ The overall mission terminates when both processes terminate (by means of the corresponding end events).

2.2 X-KLAIM

KLAIM (Kernel Language for Agents Interaction and Mobility, [20]) is a formal language specially devised to design distributed applications consisting of possibly mobile software components deployed over the nodes of a network infrastructure. KLAIM, based on process calculi [38], generalizes the notion of *generative communication*, introduced by the coordination language Linda [29], to multiple distributed tuple spaces.

¹ Notably, the duplication of the Work Alone activity in the model could be avoided by resorting to a XOR gateway that joins the sequence flows outgoing from the timer event and the not join message event; the sequence flow outgoing from this gateway would be connected to the activity. However, this model would have an unstructured topology, while our translation assumes that models are well-structured diagrams (see Sect. 3).

The implementation of KLAIM consists of two main components: the Java package KLAVA and the programming language X-KLAIM. KLAVA (KLAIM in Java, introduced initially in [5]) provides the implementation of the KLAIM concepts in terms of Java classes and methods, relying on the IMC framework [6] for the communication infrastructure. Any Java object can be stored in and retrieved from a KLAVA tuple, and the implemented pattern matching mechanism keeps Java subtyping into consideration. We also developed X-KLAIM, a domain-specific language closer to KLAIM while providing typical high-level programming constructs. X-KLAIM (eXtended KLAIM) was initially introduced in [4] and reimplemented from scratch in [7] with modern technologies and frameworks. The X-KLAIM compiler translates X-KLAIM programs into Java code that uses the Java package KLAVA. The produced Java code can then be compiled and executed using the standard Java toolchain.

The new implementation of X-KLAIM [7] is based on XTEXT [3], a mainstream Eclipse framework for the development of programming languages and Domain-Specific

Languages (DSLs). `XTEXT` also provides complete IDE support based on Eclipse: editor with syntax highlighting, code completion, error reporting, and incremental building, just to mention a few. Furthermore, we used another mechanism provided by `XTEXT`, `XBASE` [25], an extensible and reusable expression language. By using `XBASE` in `X-KLAIM`, besides a rich Java-like syntax, we inherit its interoperability with Java and its type system. An `X-KLAIM` program can smoothly access any Java type and Java library available in the project's classpath. The interoperability with Java allowed us to integrate `X-KLAIM` seamlessly with the Java-ROS connector (Sect. 2.3).

In the rest of this section, we will briefly recall the main concepts of `KLAIM` as implemented in `X-KLAIM`. We will also describe the main linguistic features of `X-KLAIM` and its Eclipse-based IDE support.

A *tuple space* is a shared data repository consisting of a multiset of tuples, i.e., sequences of data items. Tuples are associatively retrieved from tuple spaces using a *pattern-matching* mechanism. Processes communicate through the *asynchronous* exchange of tuples via tuple spaces. Tuple spaces are identified through *localities*, i.e., symbolic addresses of network nodes where processes and tuples are allocated. Localities themselves can be exchanged through interprocess communication.

An `X-KLAIM` program can contain definitions of *nets*, *nodes*, and *processes*. An `X-KLAIM` network definition consists of *net* and *node* definitions as shown in the following example:

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

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 concurrent processes. 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 the following). Typical programming structures such as *if*, *while*, and OOP Java-like mechanisms, such as object creation and method invocation, are already part of `XBASE`. The syntax of `XBASE` is similar to Java, though it removes much “syntactic noise”. For example, terminating semicolons and other syntax elements like parentheses when invoking a method without arguments are optional. Moreover, `XBASE` comes with a powerful type inference

mechanism compliant with the Java type system: the programmer can avoid specifying types in declarations when they can be inferred from the context. Variable declarations start with *val* or *var* for final and non-final variables, respectively. The types of variables can be omitted if they can be inferred from the initialization expression.

Besides the Java-like syntax mentioned above, the process body can perform the typical `KLAIM` actions. 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`. Action `in(template)@nodeLocality` (respectively, `read(template)@nodeLocality`) withdraws (respectively, reads) tuples from the tuple space hosted at the (possibly remote) locality `nodeLocality`. The distinguished locality `self` refers to the current hosting node. If matching tuples are found, one is non-deterministically chosen. Otherwise, the process is blocked until a matching tuple is found.

These retrieval actions exploit templates as patterns to select tuples in tuple spaces. *Templates* are sequences of *actual* (i.e., expressions, localities, or processes) and *formal* fields (i.e., variable definitions with the keyword *var* and the variable type), where the latter are used to bind variables to retrieved values. Intuitively, a template matches against an evaluated tuple if both have the same number of fields and corresponding fields match; two values/localities match if 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 concurrent execution to the (possibly remote) node identified by `nodeLocality`. `X-KLAIM` also supports *higher-order communication* since processes can exchange code and possibly execute it.

`X-KLAIM` also provides `read_nb` and `in_nb` constructs, which allow for non-blocking `read` and `in` operations, respectively. These non-blocking operations allow processes to continue operating even if the incoming message or signal is unavailable. They return a boolean value indicating whether they succeeded in finding a matching tuple. Thus, they are meant to be used in *if* or *while* constructs as boolean expressions.

In Fig. 3, we show a simple code snippet of an `X-KLAIM` process body. The code should be easily readable by a Java programmer. We mention a few additional `X-KLAIM` syntax features to make the code more understandable. Such types as `String` and `Double` are Java types since, as mentioned above, `X-KLAIM` programs can refer directly to Java types. Similarly, `System.err.println` is the standard Java static method to print something on the screen. In the code snippets, we omitted the Java-like import statements. Here we also see the typical `KLAIM` operations, *in* and *out*, acting

```

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

```

Fig. 3 An example of X-KLAIM code.

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). X-KLAIM provides syntactic sugar for collection literals: `#[ ... ]`. In the code snippet, `strings` is inferred to be of type `List<String>`. In X-KLAIM, *lambda expressions* have the shape `[ param1, param2, ... | body ]`. The types of parameters can be omitted if they can be inferred from the context. An X-KLAIM lambda expression can be used in any Java context where a lambda expression can be used, according to the Java type system: the type of the lambda expression must match the target Java functional interface. The code snippet shows standard Java stream operations using X-KLAIM lambda expressions.

Moreover, X-KLAIM also permits adding, after a blocking operation, a *timeout* specification (in milliseconds), using the keyword `within`. The retrieval operation becomes a boolean expression with the boolean value representing the successful retrieval within the given timeout or a failure (the timeout exceeded without finding a matching tuple). Here's an example of the usage of the version with a timeout for a retrieval operation:

```

if (in(!x, !y)@self within 2000) {
    // use the retrieved values in x and y
} else {
    // do something else
}

```

The X-KLAIM compiler is wholly 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. When saving an X-KLAIM file, the Java code generation is automatically triggered, which in turn triggers the generation of Java byte code. From a single X-KLAIM program, our compiler generates several Java classes (e.g., one for a net, one for each node, and one for each process) that extend and use KLAVA classes. The relation between X-KLAIM elements and the generated Java code is handled transparently. For example, removing a process from an X-KLAIM program will automatically remove the previously generated corresponding Java class.

Finally, the X-KLAIM Eclipse support also includes the ability to directly run or debug an X-KLAIM file with dedicated context menus. There's no need to run the generated Java code manually. Debugging an X-KLAIM source code is crucial when programming distributed applications accessing remote tuple spaces. We can set a breakpoint in the X-KLAIM program, possibly based on a condition. During the execution of the corresponding generated Java code, the execution is suspended on the X-KLAIM program: we can see the current values of variables, either in the "Variables" Eclipse view or by hovering over a variable in the program. The debugging mechanisms of X-KLAIM are as powerful as Eclipse's standard Java debugging mechanism. For example, during an X-KLAIM debugging session, we can evaluate expressions on the fly.

2.3 ROS

Robotic Operating System (ROS)² is one of the most popular frameworks for writing robot software. It provides tools and libraries for simplifying the development of complex controllers while abstracting from the underlying hardware.

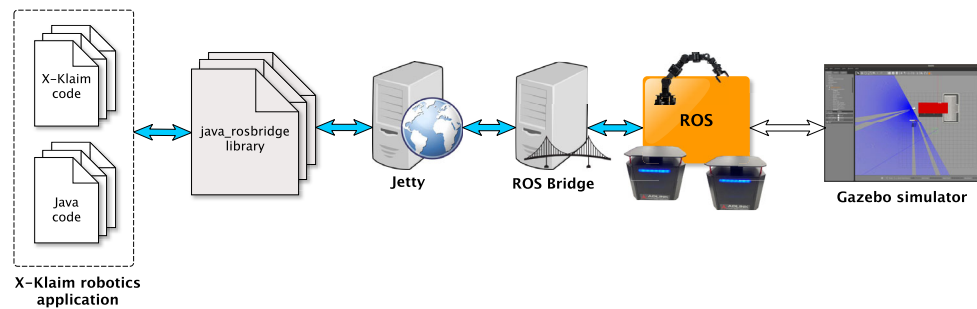
The core element of ROS is the message-passing middleware based on a publish/subscribe communication mechanism. It enables hardware abstraction for a wide variety of robotic platforms. The processes of a robotics application can exchange data, being agnostic with respect to the source of the data: the communicated data can be sensor readings or actuator commands, formatted in a standardized way, produced by, or directed to the robot's devices. To send a message, a process has to publish it in a *topic*, which is a named and typed bus. A process interested in such a message must subscribe to the topic. The subscriber will be notified whenever a new message is published on the topic. This decouples the production of data from its consumption.

To enable the use of X-KLAIM for programming the behaviors of MRS missions, we integrated in [10] the X-KLAIM language with the ROS middleware³ as graphically depicted in Fig. 4. The integration is based on the ROS Bridge, 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.

X-KLAIM programs can indirectly interact with the ROS Bridge server, publishing and subscribing over ROS topics

² <https://www.ros.org/>.

³ In this work, we used ROS 2 and Gazebo 11 classic, though our approach is version-agnostic and can be adapted to other ROS and Gazebo versions.

Fig. 4 X-KLAIM-ROS integration

via objects provided by the Java library `textitjava_rosbridge`.⁴ In its own turn, `java_rosbridge` communicates with the ROS Bridge server via the `WebSocket` protocol through the Jetty web server.⁵

ROS permits checking the execution of the code generated from an X-KLAIM program by means of the Gazebo⁶ simulator. Gazebo [35] is an open-source simulator of robot behaviors in complex environments that is based on a robust physics engine and provides a high-quality 3D visualization of simulations. Gazebo is fully integrated with ROS; in fact, ROS can interact with the simulator via the publish-subscribe communication mechanism of the framework. The use of the simulator is not mandatory when ROS is deployed in real robots. However, even in such a case, the MRS software design activity may benefit from using a simulator to save time and reduce development costs.

3 The translation process

The process of translating a BPMN model into X-KLAIM code, a vital phase of our model-driven approach, is designed to be compositional and seamless. Although it is based on an in-depth understanding of the structural and semantic nuances of both languages, the user of the approach is not required to delve into these intricacies.

In the definition of the translation, we adopted the following guiding principles. First, we aimed to ensure system integrity. The alignment of BPMN elements to their equivalent X-KLAIM constructs, in our mapping, aims to maintain the functionality and meaning of the MRS design as encapsulated within the BPMN model while transcribing it into the more execution-oriented language of X-KLAIM. In this way, the X-KLAIM program resulting from the translation reflects the same behavior as the original BPMN model. Second, we strove for accuracy and efficiency in the resulting X-KLAIM code. This ensures that the MRS will function as expected without unnecessary computational overhead. We

Table 1 Translation of sequence flows

BPMN Element	X-KLAIM Translation
	$\text{translate}(P1)$ $\text{in}(e)@\text{self}$ $\text{translate}(P2)$

also aimed at the readability and maintainability of the generated X-KLAIM programs. This means creating code that is easy to understand, update, and debug, thus improving the long-term sustainability of the MRS.

Given the diversity and complexity of BPMN elements, our systematic translation process is designed to handle these variations reliably, details of which will be discussed in the following. Notably, while some BPMN elements have a clean one-to-one mapping with X-KLAIM constructs, others require more complex translations, possibly involving multiple X-KLAIM constructs or the introduction of additional X-KLAIM code to capture the intended behavior. We define the translation for the subset of BPMN elements in Fig. 1, identified in [12] as suitable to model MRS missions; anyway, the BPMN elements discarded by this selection could be translated in X-KLAIM as well, possibly resorting to Java code integrated in X-KLAIM. Notably, we assume that BPMN models are well-structured collaboration diagrams [16]. It is worth noticing that the mapping from BPMN to X-KLAIM follows a compositional approach. This means that each BPMN construct is translated independently and then combined to form the final X-KLAIM code. This approach allows for a flexible and easy-to-modify translation, as modifications to one construct do not impact the translation of the others.

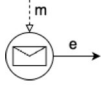
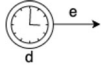
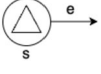
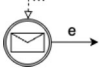
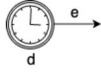
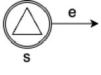
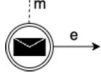
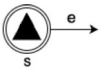
The working principles of our approach necessitate the translation of each BPMN element to dispatch a tuple representing a token marking an edge outgoing from the element before transitioning to the next element. Consumption of this tuple is essential to ensure process synchronization and execution order, as shown by the translation of the **sequence flow** provided in Table 1. This translation ensures that P1 is terminated before proceeding with P2 using the **in** action that waits for the tuple corresponding to the edge connecting

⁴ https://github.com/h2r/java_rosbridge.

⁵ Jetty 9: <https://www.eclipse.org/jetty/>.

⁶ <https://gazebo.org/>.

Table 2 Translation of events

BPMN Element		X-KLAIM Translation
None Start		out(e)@self
Message Start		in(m,vars)@self out(e)@self
Timer Start		Thread.sleep(d) out(e)@self
Signal Start		read(s,vars)@senderLoc out(e)@self
Message Intermediate Catching		in(m,vars)@self out(e)@self
Timer Intermediate Catching		Thread.sleep(d) out(e)@self
Signal Intermediate Catching		read(s,vars)@senderLoc out(e)@self
Message Intermediate Throwing		out(m,payload)@receiverLoc out(e)@self
Signal Intermediate Throwing		out(s,payload)@self Thread.sleep(signalDuration) in(s,payload)@self out(e)@self
None End		
Message End		out(m,payload)@receiverLoc
Signal End		out(s,payload)@self Thread.sleep(signalDuration) in(s,payload)@self

the two elements; such a tuple is emitted by an **out** action within the translation of P1 (see Tables 2-4).

The mapping of **events** to their X-KLAIM equivalents is reported in Table 2. Events are mainly used to deal with communications. In X-KLAIM, communication events are rendered using communication actions (i.e., **out**, **in**, and **read**); hence, both messages and signals exchanged by events are represented as tuples that embody the transmitted data.

Effective management of unicast and broadcast communication significantly aids in the seamless translation of mes-

sage and signal events, respectively. Unicast communication is characterized by message throwing/catching events within BPMN. In X-KLAIM, unicast communication is enacted through a pair of actions: the sending of an outgoing message, expressed as **out(m,payload)@receiverLoc**, originating from the current process and directed to a specific recipient, followed by the consumption of the message at the recipient's location, expressed as **in(m,vars)@self**. Notably, the two actions contain the placeholders *payload* and *vars* that will be filled by the developer with actual data of the message payload in the sent tuple and the corresponding variables in the receiving template. Instead, the location *receiverLoc* is properly determined by the translation approach, as it simply corresponds to the pool's name containing the receiving element connected to the message flow *m*. Executing these two communication actions ensures that the one-to-one communication setup inherent in unicast scenarios is accurately maintained, mimicking the precise message-passing semantics of BPMN. Broadcast communication, on the other hand, operates on a one-to-many interaction model according to the semantics of BPMN signal throwing/catching events. X-KLAIM represents broadcast communication through the emission of the signal into the local tuple space, represented as **out(s,payload)@self**, followed by the reception of this signal by multiple recipients, represented as **read(s,vars)@senderLoc**. The **read** operation permits multiple recipients to access the signal concurrently without removing it from the tuple space, facilitating the multiple-recipient scenario inherent in broadcast communication. The tuple representing the signal is then removed (via an **in** action) when a given timeout (specified by the programmer via the constant variable *signalDuration*) expires. Timer events, instead, are simply dealt with via a timeout whose duration (*d*) is specified in the BPMN model (up to a conversion in milliseconds).

The translation of **gateways** is reported in Table 3. X-KLAIM interprets the exclusive gateway through an **if-else** statement, determining a path based on conditional logic and executing the BPMN element linked with the selected path. Notably, in the generated code, *condition* is a placeholder to be replaced by the actual condition. The translation of loops, which allow repeated execution of BPMN elements, is simply based on the **while** construct. Instead, the parallel gateway initiates simultaneous execution of multiple paths (via **eval** actions), requiring the completion of all paths (see the sequence of **in** actions) before the parallel block can exit. The event-based gateway is translated differently on the basis of the connected elements. In case the gateway connects only message events, a polling strategy is implemented to determine which one is the first received message and, hence, activate the corresponding execution flow. This is achieved by means of a loop of **in** actions, one for each message, that specify the time (by means of constant

Table 3 Translation of gateways

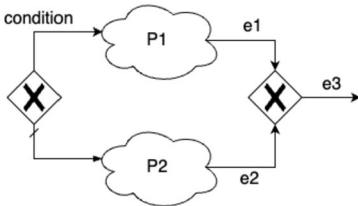
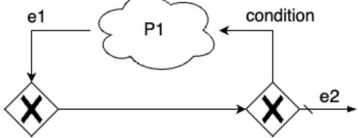
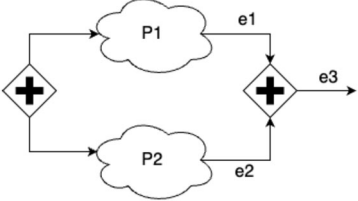
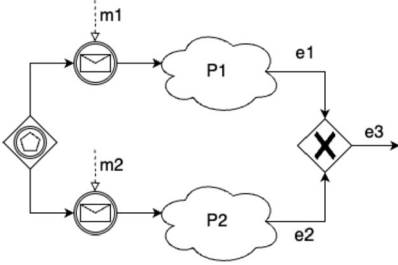
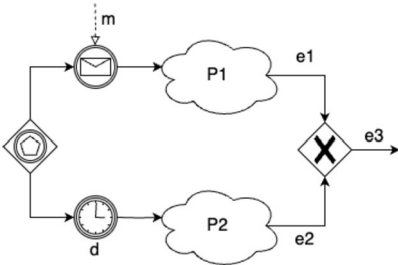
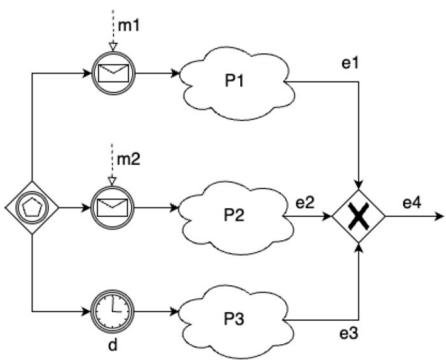
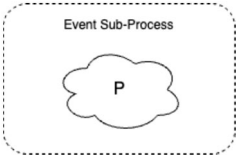
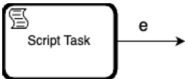
BPMN Element	X-KLA1M Translation
XOR 	<pre> if(condition){ translate(P1) in(e1)@self } else{ translate(P2) in(e2)@self } out(e3)@self </pre>
Loop 	<pre> while(condition){ translate(P1) in(e1)@self } out(e2)@self </pre>
AND 	<pre> eval(new ProcP1())@self eval(new ProcP2())@self in(e1)@self in(e2)@self out(e3)@self </pre> // Processes to be // added to the node <pre> proc ProcP1(){ translate(P1) } proc ProcP2(){ translate(P2) } </pre>
Event-Based (between messages) 	<pre> var eventOccured = false while(!eventOccured){ if(in(m1,vars1)@self within pollTimeout){ eventOccured = true translate(P1) in(e1)@self } else if(in(m2,vars2)@self within pollTimeout){ eventOccured = true translate(P2) in(e2)@self } } out(e3)@self </pre>
Event-Based (single message with timer) 	<pre> if(in(m,vars)@self within d){ translate(P1) in(e1)@self } else { translate(P2) in(e2)@self } out(e3)@self </pre>
Event-Based (between messages with timer) 	<pre> var eventOccured = false var startTime = System.currentTimeMillis() while(!eventOccured){ if(in(m1,vars1)@self within pollTimeout){ eventOccured = true translate(P1) in(e1)@self } else if(in(m2,vars2)@self within pollTimeout){ eventOccured = true translate(P2) in(e2)@self } else if(System.currentTimeMillis()-startTime >= d){ eventOccured = true translate(P3) in(e3)@self } } out(e4)@self </pre>

Table 4 Translation of activities

BPMN Element	X-K _{LAIM} Translation		
Call Activity		<code>eval(new Activity(e))@self</code>	<pre>// Process to be added to the node proc Activity(String edgeOut){ translate(CalledProcess) out(edgeOut)@self }</pre>
Event Sub-Process		<code>eval(new EventSubProcess())@self</code>	<pre>// Process to be added to the node proc EventSubProcess(){ translate(P) }</pre>
Script Task		<code>eval(new ScriptTask(e))@self</code>	<pre>// Process to be added to the node proc ScriptTask(String edgeOut){ // ... add here the script code ... out(edgeOut)@self }</pre>

pollTimeout) each message is waited for before passing to the next one. When an **in** completes successfully, the corresponding execution flow is activated and the loop is broken. The case of an event-based gateway connecting one message event and one timer is simply translated in terms of a conditional choice based on an **in** action with a timeout. Finally, the case where the event-based gateway connects more than one message, and a timer is translated by resorting again to the polling strategy; this time, at each iteration of the loop, it is checked if the timer has expired by exploiting the standard Java method `System.currentTimeMillis()`.

The translation of **activities** is shown in Table 4. Call activities and event sub-processes are translated into new processes that are defined separately. These processes are spawned using an **eval** action. The script task, instead, is translated into a placeholder for the script code, followed by an **out** action to mimic the marking of the outgoing edge at the end of the activity.

A **data object** of type `DataObject` is mapped to a tuple (`DataObject, vars`) containing variables representing the attributes of the data object. In this way, data elements can be easily managed and passed as parameters in communication actions between processes (see Tables 2-3).

Finally, the translation of **pools** is depicted in Table 5. A single-instance pool is translated into an X-K_{LAIM} node containing the translated process. The translation of a multi-instance pool, instead, involves creating multiple nodes, one for each instance (since we are considering robotics scenarios, the number of instances is determined a priori). All nodes must be then enclosed within the X-K_{LAIM} network construct **net** `CollaborationNameNet { ... }`, where `CollaborationName` is the name of the translated BPMN collaboration diagram. The translation of pools highlights the ability of X-

K_{LAIM} to handle complex collaborative processes involving multiple participants.

We conclude the section by reporting in Fig. 5 the translation into X-K_{LAIM} of the running scenario introduced in Sect. 2.1. For the sake of readability, in the following code and the ones in the subsequent sections, we omit the double quotes of constant strings that correspond to labels reported in the BPMN model (e.g., we write **out**(e1)@self in place of **out**("e1")@self).

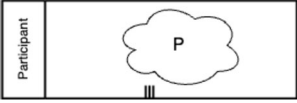
It is worth noticing that the above code contains some unnecessary synchronizations (i.e., sequences of **out** and **in** actions with the same tuple as argument), due to the compositional nature of the sequence flow translation. To achieve a cleaner and more efficient code, these redundant sequences are automatically removed by the implementation of the translation approach presented in the next sections.

4 The translation at work on an e-agriculture case study

This section explores the translation into X-K_{LAIM} of an e-agricultural MRS scenario specified in BPMN, borrowed from [12]. This scenario refers to a coordinated mission involving a drone and a tractor, orchestrated to identify and mitigate weed infestation within a crop field. Figure 6 shows a frame of the enactment of this mission; in essence, this figure is a screenshot of the execution of the generated X-K_{LAIM} code within the Gazebo simulator (we refer to Sect. 5 for a deeper discussion on code execution).

The BPMN collaboration diagram in Fig. 7 represents the overall mission behavior. It describes the workflow starting from the drone's launch, through its exploration phase, to the mission's conclusion with the identified weed removed by

Table 5 Translation of pools

BPMN Element		X-KLAIM Translation
Pool		node Participant { translate(P) }
Multi-Instance Pool		node Participant_1 { translate(P) } ... node Participant_N { translate(P) }

```

net RunningExampleNet {
  node Explorer {
    // Start event
    out(e1)@self
    in(e1)@self
    // Explore call activity
    eval(new Explore(e2))@self
    in(e2)@self
    // Message intermediate throwing event
    out(position,payload)@Worker
    out(e3)@self
    in(e3)@self
    // Event-based gateway
    var eventOccured = false
    val pollTimeout = 1000
    var startTime = System.currentTimeMillis()
    while(!eventOccured){
      if(System.currentTimeMillis()-startTime >= 30000){
        eventOccured = true
        // Work Alone call activity
        eval(new WorkAlone(e4))@self
        in(e4)@self
      } else if(in(notJoinNotification,vars1)@self within pollTimeout){
        eventOccured = true
        // Work Alone call activity
        eval(new WorkAlone(e5))@self
        in(e5)@self
      } else if(in(joinNotification,vars2)@self within pollTimeout){
        eventOccured = true
        // Message intermediate catching event
        in(arrivalNotification,vars3)@self
        out(e6)@self
        in(e6)@self
        // Work Together call activity
        eval(new WorkTogether(e7))@self
        in(e7)@self
      }
    }
    out(e8)@self
    in(e8)@self
    // End event
  }
}

```

```

proc Explore(String edgeOut){ ... out(edgeOut)@self }
proc WorkAlone(String edgeOut){ ... out(edgeOut)@self }
proc WorkTogether(String edgeOut){ ... out(edgeOut)@self }

```

```

node Worker {
  // Message start event
  in(position,vars1)@self
  out(e9)@self
  in(e9)@self
  // Evaluate position script task
  eval(new EvaluatePosition(e10))@self
  in(e10)@self
  if(isReachable){
    // Message intermediate throwing event
    out(joinNotification,payload1)@Explorer
    out(e11)@self
    in(e11)@self
    // Move to script task
    eval(new MoveTo(e12))@self
    in(e12)@self
    // Message intermediate throwing event
    out(arrivalNotification,payload2)@Explorer
    out(e13)@self
    in(e13)@self
    // Work Together call activity
    eval(new HelpTheExplorer(e14))@self
    in(e14)@self
  } else {
    // Message intermediate throwing event
    out(notJoinNotification,payload3)@Explorer
    out(e15)@self
    in(e15)@self
  }
  out(e16)@self
  in(e16)@self
  // End event
}

```

```

proc EvaluatePosition(String edgeOut){
  // ... script code ...
  out(edgeOut)@self }
proc MoveTo(String edgeOut){
  // ... script code ...
  out(edgeOut)@self }
proc HelpTheExplorer(String edgeOut){ ... out(edgeOut)@self }

```

Fig. 5 The translation into X-KLAIM of the running scenario introduced in Sect. 2.1

Fig. 6 Overview of the e-agriculture scenario

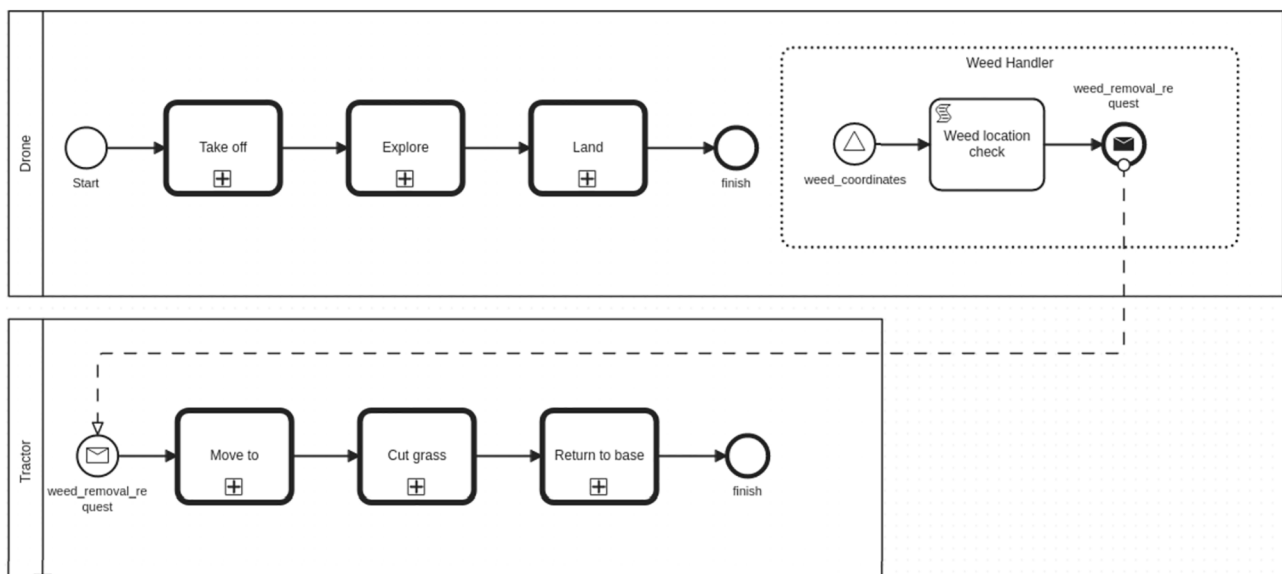
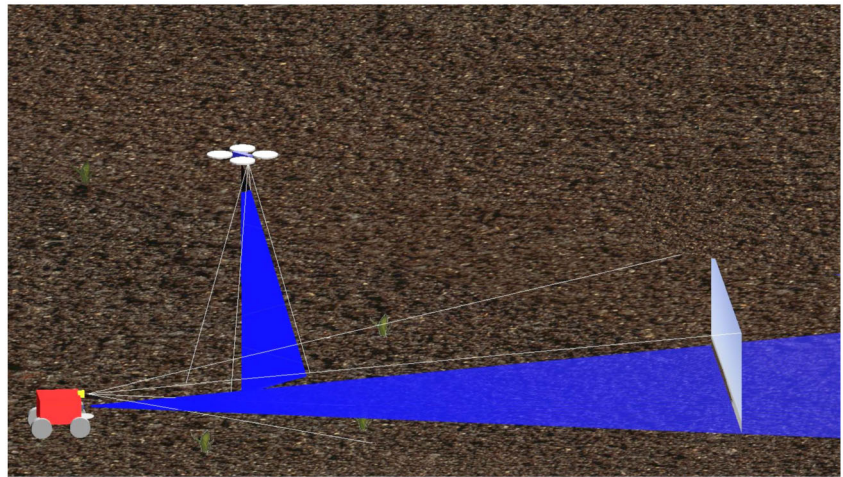


Fig. 7 BPMN model of the e-agriculture scenario

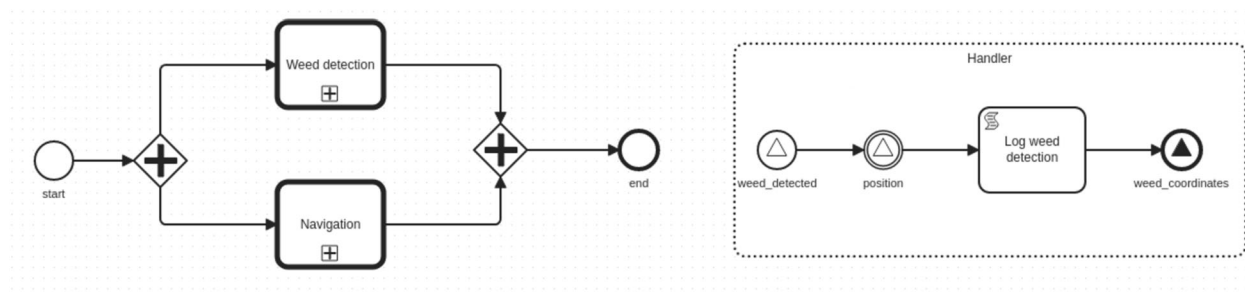


Fig. 8 BPMN model of the exploration phase

the tractor and the return of the robots to the base. The exploration phase, detailed in Fig. 8, features two parallel processes (left-hand side of the figure): one enacting a zigzag navigation strategy for a thorough field coverage and

another one for identifying the infestations. In addition, the exploration phase resorts to an event sub-process (right-hand side of Fig. 8) that deals with the precise localization of the detected weeds. The coordinates of the detected weeds are

then forwarded to the top-level model and will be caught by the event sub-process WeedHandler, which validates and communicates these locations to the tractor.

We illustrate in the following paragraphs the translation of the e-agriculture BPMN model, each targeting a specific aspect of the scenario to convey the systematic conversion into X-KLAIM. The full X-KLAIM program resulting from the translation is available at [11].

Translation of the collaboration. The collaboration within the agriculture scenario involves interactions between the drone and the tractor. In X-KLAIM, this is represented as individual nodes for Drone and Tractor, each one executing its respective behavior. The X-KLAIM code corresponding to the collaboration in Fig. 7 is as follows:

```
net MRS {
  node Drone { eval(new DroneBehavior(Tractor))@self }
  node Tractor { eval(new TractorBehavior())@self }
}
```

Translation of the Drone mission. The drone mission distills its operations into three key processes: TakeOff, Explore, and Land. This sequence, structured through event and call activities in BPMN translates into the DroneBehavior process in X-KLAIM, where each task is methodically represented. An event sub-process WeedHandler, running in parallel, manages the communication of detected weed locations, showing the drone's dual functionality in navigation and data processing. The generated code is as follows:

```
proc DroneBehavior(Locality Tractor) {
  // Initiate event subprocesses for weed handling
  eval(new WeedHandler(Tractor))@self

  // Start the main mission

  // Drone takes off
  eval(new TakeOff(e1))@self
  in(e1)@self

  // Drone explores field after takeoff
  eval(new Explore(e2))@self
  in(e2)@self

  // Drone lands after exploration
  eval(new Land(e3))@self
  in(e3)@self
}
```

Translation of the Tractor mission. The translation of the tractor mission outlines the tractor's role in the e-agricultural framework, converting its operations into sequential processes: receiving weed locations (WEED_POSITION) from the drone, moving to these locations (MoveTo), cutting the detected weeds (CutGrass), and finally returning to the base (ReturnToBase). The TractorBehavior process in X-KLAIM mirrors this workflow, depicting a direct response

to communications from the drone, thus ensuring a coherent and concise representation of the tractor's mission:

```
proc TractorBehavior() {
  // Tractor receives weeds' position from Drone
  in(WEED_POSITION, var Double x, var Double y)@self

  // Moves towards the weed position
  eval(new MoveTo(e4, x, y))@self
  in(e4)@self

  // Tractor cuts the grass after moving to the weed position
  eval(new CutGrass(e5))@self
  in(e5)@self

  // Tractor returns to the base ends after cutting the grass
  eval(new ReturnToBase(e6))@self
  in(e6)@self
}
```

Translation of the Explore process. The translation of the Explore call activity captures the drone's dual activities of field navigation and weed detection, which are crucial for e-agricultural monitoring. To this aim, it involves parallel execution of Navigation and WeedDetection processes. The event sub-process Handler, running in parallel as well, manages the precise localization of detected weeds.

```
proc Explore() {
  // Initiate event subprocesses for localizing the weed
  eval(new Handler())@self

  // Execute the two processes in parallel
  eval(new Navigation(e1))@self
  eval(new WeedDetection(e2))@self
  in(e1)@self
  in(e2)@self
}
```

5 The B2XKlaim tool

B2XKlaim is a tool designed to automate the conversion of BPMN models into X-KLAIM code. In contrast to the previous version, which required users to launch separate services, the tool is now fully deployed as a comprehensive web-based application,⁷ where both front-end and back-end components are integrated on the server, providing a seamless experience. The open source code is also available at GitHub.⁸

Backend Architecture. The B2XKlaim tool's back-end architecture is designed to efficiently translate BPMN models into X-KLAIM code. Built on top of Java and Spring Boot,

⁷ <https://kbourr.com/b2xklaim>.

⁸ <https://github.com/khalidbourr/B2XKlaim>.

this system exposes a REST API endpoint that processes BPMN XML through a structured translation pipeline.

The process begins by parsing the XML into a Document Object Model (DOM), followed by a factory pattern implementation. The `BpmnParser` class converts BPMN XML files into structured `BpmnElements` objects, each representing a BPMN element with essential contextual information. These objects are then stored in a central container for efficient lookup during subsequent translation phases. A key improvement over the previous version is the implementation of a dedicated analysis phase that follows the parsing stage. This phase carefully examines the parsed model to identify runtime dependencies between participants. This critical step analyzes Message Flows and Signal interactions across participant boundaries, building a dependency map (`requiredParticipantRefs`) that records which participant requires runtime location references to enable targeted communication.

The code translation mechanism is then facilitated by the `Generator` class, which serves as the central coordinator converting BPMN elements into X-KLAIM code. To handle diverse parsed elements (e.g., activities, events, gateways, etc.), the tool employs the `BPMNTranslator` class, which systematically processes each component according to its unique characteristics. The translation produces parametrized `eval` calls with participant arguments and corresponding `proc` definitions with `Locality` parameters, ensuring proper runtime location binding for effective communication in distributed multi-agent systems.

The translation process emphasizes code optimization in the X-KLAIM representation. While mapping BPMN elements to X-KLAIM constructs, the system enhances computational efficiency by eliminating redundant sequences. For instance, removing unnecessary `out/in` operation pairs using the same tuple reduces computational overhead while improving code readability and maintainability. This optimization serves dual purposes: streamlining execution and enhancing overall code quality.

Frontend User Interface. The user interface, developed with Vue.js, offers an intuitive environment for users to design BPMN models and initiate the translation process. Through a simple, interactive web-based platform, users can design the BPMN model, visualize the generated X-KLAIM code, and download it for finalization and deployment. This interface significantly lowers the barrier to entry for users without deep expertise in X-KLAIM programming.

A notable enhancement to the output process is the generation of a complete, downloadable Maven project archive rather than just raw code snippets. This structured project includes the generated `.xklaim` files organized into appropriate packages (`xklaim`, `xklaim.processname`, `xklaim.-activities`), placeholder procedures for unimplemented

activities, a `pom.xml` with the necessary dependencies (X-KLAIM runtime, java-rosbridge, etc.), and standard configuration files. This comprehensive approach facilitates the immediate building and execution of the generated system skeleton.

Figure 9 shows a screenshot of the user interface. The process depicted in the figure illustrates a simple drone mission consisting of the drone taking off, exploring a field, and then returning to base. These tasks are represented by call activities in the BPMN model, while the end of the mission is represented as an end event. The bottom part of the figure shows the text area reporting the corresponding X-KLAIM code produced by the tool from the translation of the given BPMN model.

6 Use case: planetary exploration

Among the various fields in which robotics is rooted, space exploration is perhaps the one where it has had the most impact, allowing us to investigate extremely remote regions of our solar system, otherwise inaccessible. Up to NASA/JPL's "Mars 2020" mission, all instances of exploration beyond the Earth have been done via single agents, be it orbiters, impactors, landers, or rovers [18]. If it is true that most rover missions did in fact consist of two separate entities, i.e., the lander platform and the rover itself, it can be argued that the platforms were not proper autonomous agents, and no real coordination happened between these and their rovers. The first instance of a truly multi-agent system was the Perseverance rover and the helicopter Ingenuity, both part of the "Mars 2020" Mission, which landed on Mars February 18, 2021. Even though the mission plan did not include proper coordinated activities between the two platforms, the remarkable performance and lifespan of Ingenuity prompted mission controllers to make active use of the small helicopter, which became the first "Extraterrestrial Scout" in History [48], albeit with non-autonomous coordination. This family of approaches has been in development in the last two decades [36], and has seen considerable progress towards the integration of proper autonomous coordination [14].

The implementation of the BPMN and X-KLAIM pipeline presented in this work lays appropriate groundwork for developing truly multi-agent frameworks for space exploration that are at the same time flexible enough for mission control to handle. In particular, we highlight the following strong points in the context of the challenges of this peculiar field:

- **Controlling complex scenarios:** it enables managing fleets of undefined size, allowing the modeling and control of arbitrarily complex directives and behaviors, possibly either coordinated, autonomous, supervised, or a mixture of these.

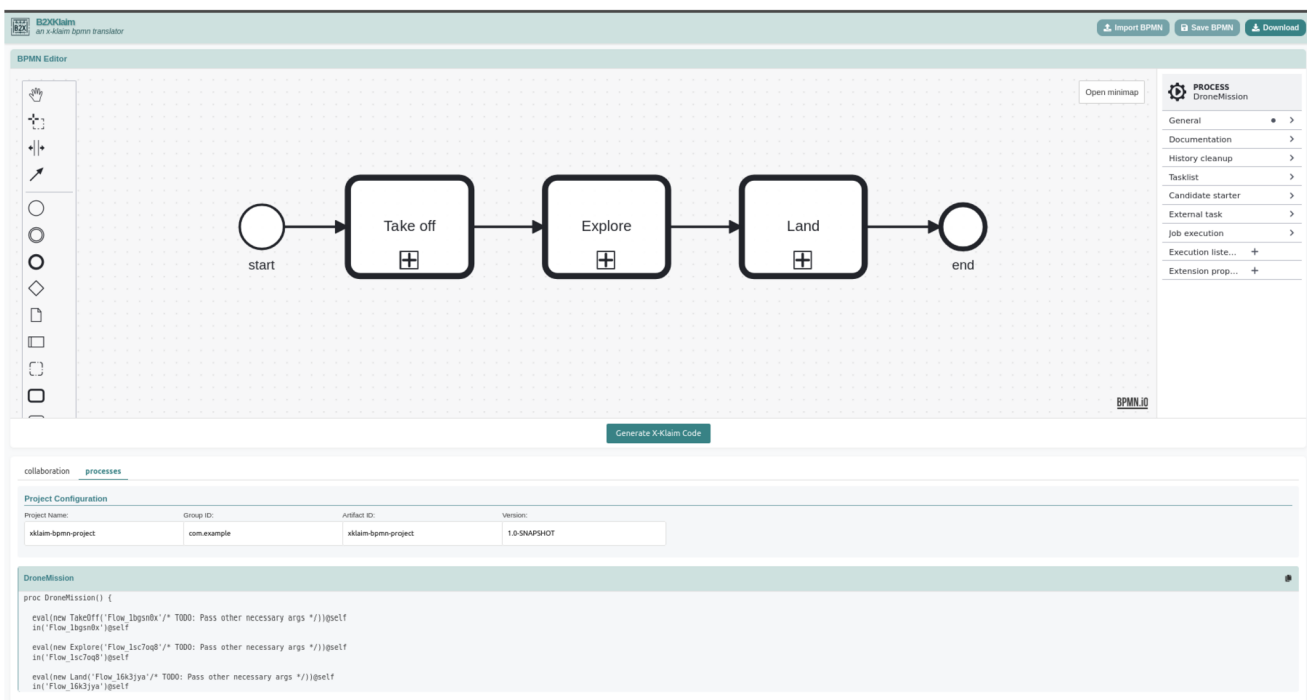


Fig. 9 B2XKlaim's user interface

- **Graphical user interface:** the BPMN convention allows users to define rules and behaviors without initially requiring in-depth knowledge of the code-base.
- **On-the-fly reconfigurability:** it enables agents to reprogram without power-cycling or software restarts, a feature vital for mission-critical infrastructure and functions (e.g., battery and power generation control systems, communication relays, heat management).

In particular, the first two points allow for the specification of missions of multi-agent fleets even to non-programmers, enabling a finer and more precise management of the endeavor directly by mission controllers on Earth. The last point is key when fleets are very heterogeneous, with radically diverse behaviours and aims; allowing for reconfiguration without creating risky downtime and restarts allows enhanced flexibility in fleet control and management.

We present, as examples, a short list of possible multi-agent mission architectures that can be modeled using the proposed framework and will constitute the backbone of our future research:

- **Helicopter scouting.** Agents: rover with scientific payload, scout helicopter; similar to the example of e-Agriculture seen in Sect. 4, this consists of an automation of the de-facto mission of the Perseverance rover aided by the Ingenuity helicopter [48];
- **Sample-collection.** Agents: sample-collection rover (SCR), rover with scientific payload (SVR), scout helicopter; this use case is a possible architecture for a sample-collection mission on Mars or on the Moon, which centers in locating a very limited number of samples with the highest possible scientific return [23]; this requires both coarse scouting (helicopter) and finer evaluation (rover), before a possible sample is deemed worthy of collection (sample-collection rover);
- **Sensor network.** Agents: carrier rovers, sensor modules; deployment of large arrays of sensor modules (e.g., for geophysical surveys, radio astronomy) via carrier rover over wide areas [14], possibly with multiple carrier rovers.
- **Lava tubes/caves exploration.** Agents: carrier rovers, cable robots, communication relays; these environments pose significant challenges due to accessibility [47] and reduced or impossible direct RF communication. Deployment and control of several agents that act as relays enables more flexibility in exploration, particularly by allowing real-time adaptation via reprogramming.

An example of a prototypical mission is depicted in Fig. 10, relative to a sample-collection mission architecture. The implementation is centered on three agents: the Sample Collection Rover (SCR), the Sample Validation Rover (SVR) and the Scout Helicopter (SH). The three agents collaborate

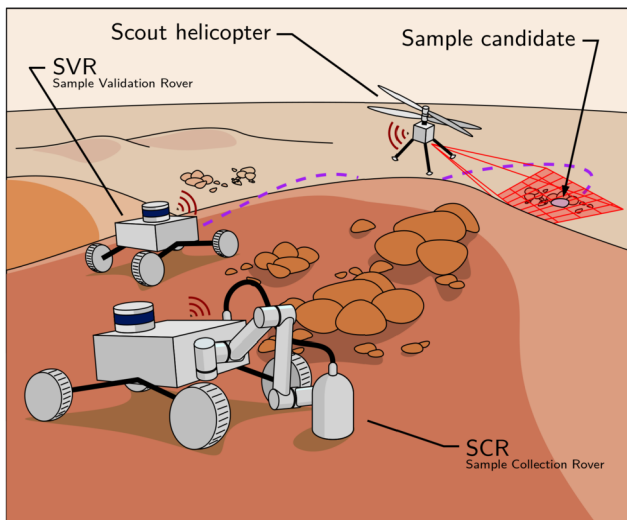


Fig. 10 Multi-agent planetary exploration scenario: three-agent sample-collection mission

to scout (SH), validate (SVR), and collect (SCR) sample candidates autonomously and—in principle—with minimal supervision from Earth's Mission Control. The robots communicate via bidirectional RF link when in line-of-sight. One of these, for example the SCR, acts as the initiator and high-level coordinator of the mission, and as such initiates the scouting by the SH. Once a candidate is detected, the SH in turn calls for the SVR to validate it; if this passes the test, the SCR receives the call to initiate its own action of sample-collection. This process is scalable in principle, and the number of any of the actors can be arbitrarily increased to meet the need of the scientific objectives of the mission.

7 Related works

The translation of BPMN into various formal and executable representations has emerged as an important research direction in both business process management and robotics domains. This section provides an overview of the existing works and positions our approach within the broader landscape of related work.

BPMN translation approaches The translation of BPMN into diverse formalisms is a growing research field, with approaches varying based on the target formalism utilized. Some studies have advocated translating BPMN diagrams to Petri Nets, thereby enabling the application of existing formal verification mechanism [31], while others have leveraged Petri Nets to develop solutions for process collaboration [46]. Significant work has also been carried out on mapping BPMN to YAWL, a language featuring strictly

defined execution semantics inspired by Petri Nets and capable of supporting verification [52]. In parallel, several studies have employed Business Process Execution Language (BPEL) as the target language, focusing on transitioning from BPMN to BPEL to expand the practical execution possibilities for BPMN diagrams [2, 22]. These approaches primarily address enterprise workflow execution rather than robotic systems coordination. A significant body of research has also utilized process calculi as a target for BPMN translation. This includes Communicating Sequential Process (CSP) based approaches, which address the challenge of BPMN verification through a mapping to CSP, enabling formal verification [50, 51]. Similarly, the π -calculus has also been adopted as a target language in the mapping [44, 45]. Other noteworthy translations include those to Concurrent Process Modeling (CPM) to facilitate the direct execution of process models [32], to Calculus for Orchestration of Web Services (COWS) for orchestrating web services [43], and to Recursive ECATNets, which can be expressed in terms of conditional rewriting logic [33].

Model-to-code translations in robotics To explore model-to-code translations within robotics, many efforts have been made to bridge the gap between high-level modeling and executable code within the robotic domain [39]. In [19], the authors have adapted BPMN for autonomous robotic tasks, employing the Trace tool to manage tasks and respond to unplanned events. Similarly, [24, 27, 37] have used Petri-Nets to model ROS nodes, enhancing the verification and control of ROS-based systems. Beyond Petri Net-based approaches, other modeling paradigms have been employed for robotic code generation. For example, [28] presented a DSL designed for specifying the kinematics and dynamics of rigid-link robots, making it feasible to generate executable code. Behavior-based modeling approaches have also gained attention; for instance, [30] developed a framework that transforms Behavior Trees into executable robot code; UML has also been used to model and design robotic tasks and missions, e.g., Art2ool [26] supports the development cycle of robotic arm tasks in which atomic tasks are abstracted with UML class diagrams. For state-based modeling, [34] proposed rFSM, a statechart-based coordination language with code generation capabilities specifically designed for robotic applications.

Differentiating our approach Our approach, mapping BPMN to X-KLAIM, distinctly sets itself apart from existing works by focusing on the coordination of heterogeneous multi-robot systems rather than limiting its scope to single ROS-based systems, as many studies do, or to homogeneous robots, as proposed in [1]. Moreover, unlike previous studies, which generally focus on formal verification of BPMN, our

method primarily aims at code generation, delivering executable (skeleton) code from BPMN collaboration diagrams. This practical emphasis facilitates a clear comprehension of participant roles and message exchanges, enhancing usability in real-world applications. Additionally, we employ a target language, X-KLAIM, grounded in the formal language KLAIM, to make the output code amenable to formal analysis, hence paving the way for future robustness and safety considerations in complex systems like autonomous robots. The compatibility with the prevalent ROS middleware and the potential adaptability to its future versions foster its applicability to modern robotics software infrastructures [8, 9]. Note that since X-KLAIM is completely interoperable with Java, it can also seamlessly access ROS through the Java ROS bridge library. In [10], we illustrated a few benchmark experiments showing that our code (X-KLAIM and Java ROS bridge) introduces a slightly more significant but acceptable latency and consumes less memory than the traditional ROS implementation based on Python code.

8 Concluding remarks

In this paper, we propose a model-driven approach to obtain executable code for a MRS from a mission designed in BPMN. Our primary contribution lies in bridging the substantial gap between high-level mission design and the practical execution of heterogeneous multi-robot systems, enhancing the development process by enabling a direct transition from conceptual models to operational code. By starting with a BPMN model, roboticists and domain experts can together design missions at a high level of abstraction, ensuring that all stakeholders clearly understand the mission's purpose and broad steps. Then, the model can be systematically translated into an X-KLAIM program using the B2XKlaim tool, preserving the structure and logic of the original design while adding the necessary detail for execution in a ROS environment. This approach is particularly effective, with respect to directly writing the code in X-KLAIM, when the design of the MRS mission involves beginner roboticists, domain experts, and other stakeholders who may not have a strong programming background and precise control over concurrent process interactions.

Since the X-KLAIM compiler generates plain Java code, which depends only on KLAIVA and a few small libraries, an X-KLAIM application can be deployed by using standard Java tools and mechanisms. For example, it is enough to create a JAR with the generated Java code and its dependencies (KLAIVA and *java_rosbridge*), that is, a so-called “fat-jar” or “uber-jar”. Such a JAR file can then be deployed to every physical robot where a Java virtual machine is already installed. 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.

It is worth noticing that the generated X-KLAIM code is meant to be modified by the user to add proper information that cannot be derived from the BPMN model. If the user modifies the BPMN model and runs the code generation again, the manual modifications to the X-KLAIM code will be lost. This is a typical problem of generated code meant to be customized. It can be handled using a version control system, like Git, to reapply the manual modifications to the newly generated code. In particular, the Eclipse Git integration, *EGit*, can be used to easily see such changes.

An example is shown in Fig. 11. In this example, we had previously generated the X-KLAIM project and applied a few manual changes to the code. We had added a few formal fields in the **in** operation. In the subsequent **eval**, we had used the values of matched formal fields and also added an additional string value. We had committed such changes in the Git repository. Later, we re-generate the code after changing the BPMN model, e.g., by increasing the timeout. Using the Git view, we can easily see all the changes with respect to the code in the Git repository. Note how the Eclipse Git integration not only highlights the changed parts, but also highlights the single changed parts within the changed lines. The highlighted red parts concern the manual changes we had previously done, which are, of course, removed in the regenerated code. By using the toolbar buttons related to each change, we can easily go back to our manually modified code and decide to keep the changes relative to the modified BPMN model, i.e., in this example, the new timeout value (corresponding to the **sleep** statement).

Other mechanisms to deal with this situation are under investigation. For example, we could use the *Generation Gap* pattern [49], a pattern that solves the problem of maintainability of coexisting generated and manual code by relying on class inheritance: a base class encapsulates the generated code, and a derived class encapsulates the custom code. Alternatively, the custom code could be written in a separate file, not affected by the code generation. We could also extend the B2XKlaim editor to allow the developer to specify custom code that will be inserted into the generated code.

In future work, we plan to demonstrate the scalability and robustness of our approach by implementing more complex scenarios, such as those outlined in Sect. 4, or others that may utilize multi-instance pools for homogeneous robots. This will allow us to assess the usability and efficiency of our system when managing large-scale scenarios. In this respect, it is worth noticing that the scalability of the approach is mainly affected by the use of a graphical notation for designing the robotic system interactions

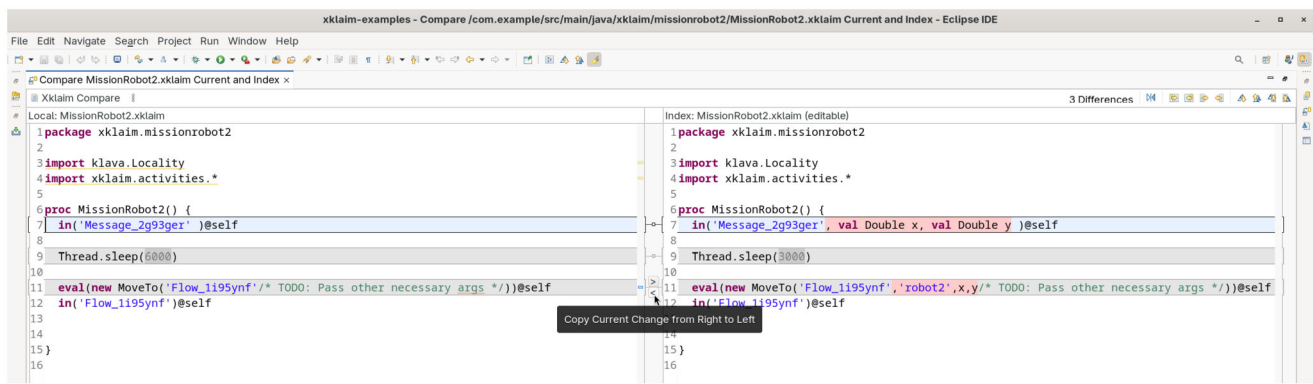


Fig. 11 The Eclipse Git view showing the locally modified code (left) and the version in the Git repository (right)

rather than by the translation process we propose. As mentioned in Sect. 2.1, BPMN offers constructs and mechanisms (e.g., subprocesses, multi-instance pools, call activities) to deal with large models. However, using BPMN may still become unfriendly when modeling particularly complex scenarios. Comprehensive evaluations will be conducted, including case studies and empirical evidence, to substantiate the benefits and limits of our approach in varied robotics environments. Additionally, we plan to explore integrating our approach with real-time safety systems to handle unforeseen scenarios (e.g., human presence detection, emergency stops, and obstacle avoidance). This could leverage BPMN constructs, such as event sub-processes that can model such contingencies, coupled with runtime adaptation mechanisms at the level of X-KLAIM and/or ROS.

Our long-term goal is to implement an MRS that exploits our approach to enable real-world missions in the space exploration, search-and-rescue, and industrial domains, under conditions closely resembling real operations.

Additionally, we plan to further optimize the X-KLAIM code generated by the B2XKlaim translator. In particular, in addition to the code optimizations discussed in Sect. 5, we could replace some redundant synchronizations, resulting e.g. from the translation of task sequences, by using the X-KLAIM primitive `executeNodeProcess`, which performs a synchronous call of a process.

Funding information This work was partially funded by the European Union - Next Generation EU, Mission 4 Component 1, CUP J53D23007120001, project AI4FOREST. The work was also partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - Next Generation EU.

References

- Aguzzi, G., Audrito, G., Casadei, R., Damiani, F., Torta, G., Viroli, M.: A field-based computing approach to sensing-driven clustering in robot swarms. *Swarm Intell.* **17**(1), 27–62 (2023)
- Bai, L., Wei, J.: A service-oriented business process modeling methodology and implementation. In: *International Conference on Interoperability for Enterprise Software and Applications*, pp. 201–205. IEEE Press, New York (2009)
- Bettini, L.: *Implementing Domain-Specific Languages with Xtext and Xtend*, 2nd edn. Packt Publishing (2016)
- Bettini, L., De Nicola, R., Pugliese, R., Ferrari, G.L.: Interactive mobile agents in X-Klaim. In: *WETICE*, pp. 110–117. IEEE Comput. Soc., Los Alamitos (1998)
- Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java package for distributed and mobile applications. *Softw. Pract. Exp.* **32**(14), 1365–1394 (2002)
- Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A flexible and modular framework for implementing infrastructures for global computing. In: *DAIS, LNCS*, vol. 3543, pp. 181–193. Springer, Berlin (2005)
- Bettini, L., Merelli, E., Tiezzi, F.: X-Klaim is back. In: *Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS*, vol. 11665, pp. 115–135. Springer, Berlin (2019)
- Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Writing robotics applications with X-Klaim. In: *ISoLA, LNCS*, vol. 12477, pp. 361–379. Springer, Berlin (2020)
- Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Programming multi-robot systems with X-KLAIM. In: *ISoLA, LNCS*, vol. 13703, pp. 283–300. Springer, Berlin (2022)
- Bettini, L., Bourr, K., Pugliese, R., Tiezzi, F.: Coordinating and programming multiple ROS-based robots with X-KLAIM. *Int. J. Softw. Tools Technol. Transf.*, 1–18 (2023)
- Bourr, K.: B2XKLAIM repository. <https://github.com/khalidbourr/B2xklaim-e-agriculture>
- Bourr, K., Corradini, F., Pettinari, S., Re, B., Rossi, L., Tiezzi, F.: Disciplined use of BPMN for mission modeling of multi-robot systems. In: *PoEM-Forum, CEUR Workshop Proceedings*, vol. 3045, pp. 1–10. CEUR-WS.org (2021)
- Bourr, K., Tiezzi, F., Bettini, L.: Model-driven development of multi-robot systems: from BPMN models to x-klaim code. In: *ISoLA, LNCS*, vol. 15220, pp. 224–242. Springer, Berlin (2024)
- Burkhard, L., Sakagami, R., Lakatos, K., Gmeiner, H., Lehner, P., Reill, J., Muller, M.G., Durner, M., Wedler, A.: Collaborative multi-rover crater exploration: concept and results from the ARCHES analog mission. In: *IEEE Aerospace Conference Proceedings*, pp. 1–14. IEEE Press, New York (2024)
- Compagnucci, I., Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F.: A systematic literature review on iot-aware business process modeling views, requirements and notations. *Softw. Syst. Model.* **22**(3), 969–1004 (2023)
- Corradini, F., Morichetta, A., Muzi, C., Re, B., Tiezzi, F.: Well-structuredness, safeness and soundness: a formal classification

- of bpmn collaborations. *J. Log. Algebr. Methods Program.* **119**, 100630 (2021)
17. Corradini, F., Pettinari, S., Re, B., Rossi, L., Tiezzi, F.: A BPMN-driven framework for multi-robot system development. *Robot. Auton. Syst.* **160**, 104322 (2023)
 18. Cottiga, S., Caruso, M., Gallina, P., Seriani, S.: Proprioceptive swarms for celestial body exploration. *Acta Astronaut.* **223**, 159–174 (2024)
 19. de la Croix, J.P., Lim, G.: Event-driven modeling and execution of robotic activities and contingencies in the Europa lander mission concept using BPMN. In: *International Symposium on Artificial Intelligence, Robotics and Automation in Space* (2020)
 20. 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)
 21. Dhoub, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: *SIMPAP, LNCS*, vol. 7628, pp. 149–160. Springer, Berlin (2012)
 22. Dubani, Z., Soh, B., Seeling, C.: A novel design framework for business process modelling in automotive industry. In: *International Symposium on Electronic Design, Test & Applications*, pp. 250–255. IEEE Press, New York (2010)
 23. Durner, M., Boerdijk, W., Fanger, Y., Sakagami, R., Risch, D.L., Triebel, R., Wedler, A.: Autonomous rock instance segmentation for extra-terrestrial robotic missions. In: *IEEE Aerospace Conference Proceedings*, pp. 1–14. IEEE Press, New York (2023)
 24. Ebert, S., Mey, J., Schöne, R., Götz, S., Abmann, U.: DiNeROS: a model-driven framework for verifiable ROS applications with Petri nets. In: *MODELS*, pp. 791–800. IEEE Press, New York (2023)
 25. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. In: *GPCE*, pp. 112–121. ACM, New York (2012)
 26. Estévez, E., García, A.S., García, J.G., Ortega, J.G.: Art2ool: a model-driven framework to generate target code for robot handling tasks. *Int. J. Adv. Manuf. Technol.* **97**(1–4), 1195–1207 (2018)
 27. Figat, M., Zieliński, C.: Robotic system specification methodology based on hierarchical Petri nets. *IEEE Access* **8**, 71617–71627 (2020)
 28. Frigerio, M., Buchli, J., Caldwell, D.G.: A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In: *DSLRob* (2013). [arXiv:1301.7190](https://arxiv.org/abs/1301.7190)
 29. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
 30. Ghzouli, R., Berger, T., Johnsen, E.B., Wasowski, A., Dragule, S.: Behavior trees and state machines in robotics applications. *IEEE Trans. Softw. Eng.* **49**(9), 4243–4267 (2023)
 31. Huai, W., Liu, X., Sun, H.: Towards trustworthy composite service through business process model verification. In: *International Conference on Ubiquitous Intelligence & Computing and International Conference on Autonomic & Trusted Computing*, pp. 422–427. IEEE Press, New York (2010)
 32. Ji, Y., Sun, H., Liu, X., Zeng, J., Bai, S.: A decentralized framework for executing composite services based on BPMN. In: *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 332–338. IEEE Press, New York (2009)
 33. Kheldoun, A., Barkaoui, K., Ioualalen, M.: Specification and verification of complex business processes—a high-level Petri net-based approach. In: *BPM, LNCS*, vol. 9253, pp. 55–71. Springer, Berlin (2015)
 34. Klotzbücher, M., Bruyninckx, H.: Coordinating robotic tasks and systems with rfsm statecharts. *J. Softw. Eng. Robot.* **3**(1), 28–56 (2012)
 35. Koenig, N.P., Howard, A.: Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: *IROS*, pp. 2149–2154. IEEE Press, New York (2004)
 36. Lindemann, R.A., Bickler, D.B., Harrington, B.D., Ortiz, G.M., Voorhees, C.J.: Mars exploration rover mobility development - mechanical mobility hardware design, development, and testing. *IEEE Robot. Autom. Mag.* **13**(2), 19–26 (2006)
 37. López, J., Sánchez-Vilariño, P., Sanz, R., Paz, E.: Implementing autonomous driving behaviors using a message driven Petri net framework. *Sensors* **20**(2), 449 (2020)
 38. Milner, R.: *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, New York (1989)
 39. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: *SIMPAP, LNCS*, vol. 8810, pp. 195–206. Springer, Berlin (2014)
 40. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. *Softw. Eng. Robot.* **7**, 75–99 (2016)
 41. Omg, O., Parida, R., Mahapatra, S.: Business process model and notation (bpmn) version 2.0. *Object Manag. Group* **1**(4), 18 (2011)
 42. Open Robotics: Robot Operating System (ROS) website. <https://www.ros.org/>
 43. Prandi, D., Quaglia, P., Zannone, N.: Formal analysis of BPMN via a translation into COWS. In: *COORDINATION, LNCS*, vol. 5052, pp. 249–263. Springer, Berlin (2008)
 44. Puhlmann, F.: Soundness verification of business processes specified in the pi-calculus. In: *On the Move to Meaningful Internet Systems*, vol. 4803, pp. 6–23. Springer, Berlin (2007)
 45. Puhlmann, F., Weske, M.: Investigations on soundness regarding lazy activities. *Bus. Process Manag.* **4102**, 145–160 (2006)
 46. Roa, J., Chiotti, O., Villarreal, P.: A verification method for collaborative business processes. In: *BPM, LNBIP*, vol. 99, pp. 293–305. Springer, Berlin (2012)
 47. Seriani, S., Gallina, P., Wedler, A.: A modular cable robot for inspection and light manipulation on celestial bodies. *Acta Astronaut.* **123**, 145–153 (2016)
 48. Tzanetos, T., Aung, M., Balaram, J., Grip, H.F., Karras, J.T., Canham, T.K., Kubiak, G., Anderson, J., Merewether, G., Starch, M., Pauken, M., Cappucci, S., Chase, M., Golombek, M., Toupet, O., Smart, M.C., Dawson, S., Ramirez, E.B., Lam, J., Stern, R., Chahat, N., Ravich, J., Hogg, R., Pipenberg, B., Keenon, M., Williford, K.H.: Ingenuity Mars helicopter: from technology demonstration to extraterrestrial scout. In: *IEEE Aerospace Conference Proceedings*, pp. 01–19. IEEE Press, New York (2022)
 49. Vliissides, J.: *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, Reading (1998)
 50. Wong, P.Y., Gibbons, J.: A process semantics for BPMN. In: *ICFEM, LNCS*, vol. 5256, pp. 355–374. Springer, Berlin (2008)
 51. Wong, P.Y., Gibbons, J.: Formalisations and applications of BPMN. *Sci. Comput. Program.* **76**(8), 633–650 (2011)
 52. Wynn, M.T., Verbeek, H., van der Aalst, W.M., ter Hofstede, A.H., Edmond, D.: Business process verification—finally a reality! *Bus. Process Manag. J.* **15**(1), 74–92 (2009)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.