Università degli Studi di Camerino

School of Advanced Studies

Dottorato di Ricerca in Scienze e Tecnologie
Computer Science - XXXV Ciclo

DSLs for Modelling, Coordinating and Programming Multi-Robot Systems

Relatore

Prof. Francesco TIEZZI

Dottorando

Khalid BOURR

Commissione Esaminatrice

Prof. Alberto Lluch LAFUENTE

Prof. Stefano SERIANI

Anno Accademico 2022-2023

University of Camerino

School of Advanced Studies

Doctor of Philosophy in Sciences and Technology

Computer Science - XXXII Cycle

DSLs for Modelling, Coordinating and Programming Multi-Robot Systems

Supervisor

Prof. Francesco TIEZZI

PhD Candidate

Khalid BOURR

Doctoral Examination Committee

Prof. Alberto Lluch LAFUENTE

Prof. Stefano SERIANI

Academic Year 2022-2023

"You can never cross the ocean until you have the courage to lose sight of the shore."

-Christopher Columbus

Abstract of the Dissertation

Software development for robotics applications is still a major challenge that becomes even more complex when considering a Multi-Robot System (MRS). Such a distributed software has to perform multiple cooperating tasks well-coordinatedly to avoid unsatisfactory emerging behavior. This thesis presents a high-level programming and modeling approach for MRSs using X-Klaim programming language and the Business Process Modeling Notation (BPMN). The computation and communication model of X-Klaim, based on multiple distributed tuple spaces, permits coordinating with the same abstractions and mechanisms both intra- and inter-robot interactions of an MRS. This allows developers to focus on the MRS behavior, achieving readable, reusable, and maintainable code. Alongside this, BPMN offers an effective mechanism for representing the high-level design of MRS missions. Its graphical nature aids in depicting complex mission sequences and interactions in a way that is easily comprehensible, promoting clarity and interoperability. Our approach consolidates these two paradigms, enabling a smooth transition from the conceptual design to concrete implementation of MRS missions. We achieve this by providing a systematic mapping of BPMN elements to X-Klaim constructs, which preserves the logic and structure of the original design while adding necessary detail for execution in Robot Operating System (ROS) environment. ROS is a flexible framework for writing robot software and provides the tools, libraries and conventions to simplify the process. To validate the feasibility and effectiveness of our approach, we implemented and tested it across different MRS scenarios. The results showed that our approach scales effectively when applied to increasingly complex scenarios, allowing for code reusability. Further, our solution introduces a slightly higher but acceptable latency compared to traditional ROS implementations based on Python code, and it consumes less memory. This research contributes towards improving the efficiency and quality of MRS software development, bringing together the power of BPMN, X-Klaim and ROS in a synergistic and integrated design-to-implementation process. Our

findings offer promising implications for future development of more complex, scalable, and effective MRS missions.

Acknowledgments

The journey through this PhD was a series of interconnected challenges, where each experience organically led to the next, enriching my academic voyage. In Camerino, my arrival was marked by an unforeseen dark downpour. The rain soaked my phone, rendering it inoperative and thus depriving me of GPS, which left me without immediate guidance to a shelter, foreshadowing the intricate maze of challenges and twists my PhD journey was set to unveil. Moving from the structured realm of Mathematics to the expansive horizons of Computer Science was more than just an academic shift; it necessitated a deep cognitive transformation and an evolution in my approach to challenges. This academic evolution took another turn when my mentor moved to another university, compelling me to swiftly adapt to a new academic environment. Parallel to these intellectual adaptations was a cultural journey—settling in a new land, away from the familiar warmth of family, I found myself balancing the academic rigors with the nuances of an unfamiliar cultural landscape.

Yet, every challenge was surmounted, including speaking the Italian language, not just by personal grit but also due to the unwavering support and guidance I received.

At the forefront of this journey, Professor Francesco Tiezzi has been my guiding star. His unwavering support, invaluable insights, and relentless dedication have been the cornerstone of my PhD experience. Without him, the completion of this thesis would undoubtedly remain a distant dream. My heartfelt gratitude to him for every discussion, every guidance, and every moment he invested in me. In parallel, I must acknowledge Professor Barbara Re. Her assistance and insights added depth to my research, and I am deeply thankful for her contributions.

To the esteemed committee members who oversaw and critiqued my work, your contributions were invaluable in refining my research. Professor Alberto Lluch Lafuente and Professor Stefano Seriani, your expertise and feedback not only helped shape the outcome of this study but also broadened my

understanding of the subject at hand. Your constructive insights will remain a foundational part of my academic journey.

Further, the very inception of this journey owes much to the PhD admissions committee at University of Camerino. In particular, I wish to extend my gratitude to Professor Michele Loreti, whose role as the coordinator of the PhD program was pivotal in setting the direction of my research. Similarly, Professor Andrea Polini's contributions have been instrumental, and I deeply appreciate the trust and faith both of you placed in me.

Within the walls of the PROS Lab, I journeyed through vibrant landscapes of innovation and team synergy. Professor Flavio Corradini, with his dynamic leadership, transformed the lab into a beacon of knowledge and motivation. My tenure there thrives on a foundation of excellence and perpetual learning. My heartfelt gratitude goes out to Professor Corradini and the entire team for sculpting my academic trajectory.

Navigating the intricate world of academia, the collaborative spirit served as my guiding wind. My voyage was deeply enriched by the wisdom of Professors Rosario Pugliese and Lorenzo Bettini from the University of Florence, whose insights reshaped my research perspectives. Their mentorship remains a cherished chapter of my journey. Moreover, the expertise of Professors Patrizio Pellicione, Catia Trubiani, and Riccardo Pincirolli from the GSSI was instrumental. Their guidance gave my research depth and breadth, and their unwavering support transcended mere professional ties, for which I am profoundly grateful.

Throughout my journey, Dr. Fabrizio Fornari was not only a guiding figure academically but also a pillar of support beyond the confines of the university. His unwavering assistance, both in addressing bureaucratic intricacies and personal challenges, was invaluable. I am deeply grateful to him for his steadfast presence. Additionally, my thanks extend to Dr. Lorenzo Rossi and Dr. Michela Quadrini for their insights and support. Alongside them, the cameraderie of Aniqa Rahman, Alessandro Marcelleti, Caterina Luciani and Sara Pettinari was instrumental. Beyond just colleagues, they were steadfast allies and friends. Their unwavering encouragement and the moments we shared have greatly enriched my path. To all, my profound gratitude for guiding and accompanying me in this academic adventure.

Last but not least, nothing can be achieved without the unconditional support of family. My parents through every challenge and triumph, their unwavering faith and boundless love have been my guiding force. Even though miles separated us, their encouragement has always bridged every distance. My brothers, too, have added immeasurable depth to my journey with their invaluable experiences. I owe my deepest gratitude; They are the bedrock upon which all my achievements rest.

To all these luminaries, and many others who have touched my academic

journey, my sincere thanks. You have made this more than just a pursuit of knowledge; you've made it an unforgettable adventure.

This work was partially supported by the PRIN projects "SEDUCE" n. 2017TWRCNB.

List of Publications

- Bettini, Lorenzo. Bourr, Khalid. Pugliese, Rosario. Tiezzi, Francesco. "Writing robotics applications with X-Klaim." In Proceedings of 9th International Symposium on Leveraging Applications of Formal Methods Verification and Validation: Engineering Principles, ISoLA 2020, Part II (Vol. 12477, pp. 361-379). Lecture Notes In Computer Science. Springer.

- Bourr, Khalid. Corradini, Flavio. Pettinari, Sara. Re, Barbara. Rossi, Lorenzo. Tiezzi, Francesco. "Disciplined use of BPMN for mission modeling of Multi-Robot Systems." In Proceedings of the Practice of Entreprise Modeling Conference (PoEM) Forum. 2021 (Vol. 3045, pp.1-10).

- Bettini, Lorenzo. Bourr, Khalid. Pugliese, Rosario. Tiezzi, Francesco. "Programming multi-robot systems with X-Klaim." In Proceedings of 11th International Symposium on Leveraging Applications of Formal Methods Verification and Validation: Engineering Principles , ISoLA 2022, Part III (Vol. 13703, pp. 283-300). Lecture Notes In Computer Science. Springer.

- Bettini, Lorenzo. Bourr, Khalid. Pugliese, Rosario. Tiezzi, Francesco. "Coordinating and Programming Multiple ROS-Based Systems with X-Klaim." International Journal on Software Tools for Technology Transfer Journal, 2023 (to appear).

- Bourr, Khalid. Tiezzi, Francesco. "From BPMN to X-Klaim: A systematic Methodology for model translation and program generation" (ongoing work).

Contents

PART I

INTRODUCTION AND BACKGROUND

Chapter 1

Introduction

In the contemporary landscape of computing and robotics, *Multi-Robot Systems* (MRSs) have emerged as an integral aspect of a broad spectrum of applications, ranging from exploration missions in inhospitable environments to industrial automation and precision agriculture. Their widespread adoption can be attributed to the many advantages they offer, including improved performance, robustness, scalability, and fault tolerance. However, as the adoption of MRSs continues to rise, the inherent complexities associated with their programming and coordination are becoming more pronounced.

1.1 Motivations

The evolution of Multi-Robot Systems (MRSs) has been both promising and problematic. While they offer tremendous advantages in terms of performance and scalability, the complexities tied to their programming and coordination cannot be underestimated. Traditional methods have limitations that make them inefficient and prone to errors, calling for more abstract and high-level solutions. This thesis addresses these critical challenges by introducing a tailored Domain-Specific Languages (DSL) for MRS programming and coordination, and by enhancing the prototyping of MRS missions. Furthermore, it establishes a seamless transition from high-level mission models to low-level implementations. To do this effectively, the thesis leverages the computational robustness of X-Klaim and the expressive power of BPMN. X-Klaim has been chosen for its proficiency in simplifying programming tasks associated with distributed systems, such as MRSs, and BPMN, for its versatility in modeling the high-level interactions within these systems. Together, they create a comprehensive, efficient, and more understandable framework for the design and implementation of MRSs, offering concrete solutions to the motivations outlined and making significant contributions to

the field.

1.1.1 Complexity of MRS Programming

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

The development of the software controlling a single autonomous robot is still a challenge [1, 2, 3]. This becomes even more arduous in the case of MRSs [4, 5], as it requires dealing with multiple cooperating tasks to drive the robots to work as a well-coordinated team. To meet this challenge, various software libraries, tools and middlewares have been proposed to assist and simplify the rapid prototyping of robotics applications. Among them, nowadays, a prominent solution is ROS (Robot Operating System [6]), a popular framework largely used in both industry and academia for writing robot software. On the one hand, ROS provides a layer to interact with many sensors and actuators for many robots while abstracting from the underlying hardware. On the other hand, programming with ROS still requires dealing with low-level implementation details; hence, robotics software development remains a complex and demanding activity for practitioners from the robotic domain. To face this issue, many researchers have proposed using Domain-Specific Languages (DSLs) which offer higher-level abstractions tailored to a specific problem domain, thus simplifying complex programming tasks. Programmers exploit these to drive the software development process and then resort to tools for the automated generation of executable code and system configuration files. Many proposals in the literature are surveyed in [7, 3, 8, 9].

Along this line of research, we propose using the language X-KLAIM to program multiple ROS-based systems. This choice is motivated by the fact that X-KLAIM provides mechanisms based on distributed tuple spaces for coordinating the interactions between these software components at a high level of abstraction. In fact, the X-KLAIM's computation and communication model is particularly suitable for dealing both with *(i)* the distributed nature of the architecture of each robot belonging to an MRS, where the software components dealing with actuators and sensors execute concurrently, and *(ii)* the inherent distribution of the MRS, which is formed by multiple interacting robots. Notably, the same tuple-based mechanisms are used both for intra- and inter-robot communication. This simplifies the design

and implementation of MRS's software in terms of an X-Klaim application distributed across multiple threads of execution and hardware platforms, resulting in better readable, maintainable, and reusable code.

1.1.2 The Need for High-Level Modeling and Abstraction

The design and development of missions for MRSs present a complex array of challenges. These range from the coordination of tasks between robots [10], management of environmental uncertainty [11], and task allocation [12], to the need for real-time responsiveness and robustness against individual robot failure. With the increasing complexity of MRSs and the tasks they are designed to perform, traditional low-level coding methods have proven to be labor-intensive, error-prone, and inadequate for the task at hand [8, 9, 5].

A promising alternative that has been gathering momentum in recent years is the use of high-level modeling and abstraction techniques, such as the Business Process Model and Notation (BPMN) [13]. BPMN has emerged as widely adopted graphical representation for specifying business processes in a workflow. Its versatility and expressive power make it particularly suited for modeling the intricate interaction of workflow, for example in IoT [14].

BPMN facilitates the creation of intuitive, high-level models that effectively capture the collaborative behavior of MRSs. This approach not only simplifies the development process but also enhances its efficiency [15]. The high expressivity of BPMN collaborations encapsulates the interplay of control flow, data flow, and communication within a unique diagram, making the modeling activity easier to manage and understand. The resulting models are human-readable, providing a common language for stakeholders, and they are also suitable for automated model-to-code solutions, reducing potential coding errors and increasing the efficiency of the development process [16].

Furthermore, BPMN is backed by a wealth of supportive tools and techniques that enhance the development of high-quality systems. These include formalization, which aids in defining precise and unambiguous models [17], execution and animation tools, which provide the ability to simulate and validate models prior to deployment [18], and verification mechanism, which ensure that the model satisfies a set of specified properties before its implementation [19].

Importantly, in this thesis we propose the usage of the BPMN standard without extensions, demonstrating that is possible to successfully model the behavior of a cooperative MRS using only the tools and constructs defined in the BPMN 2.0 standard. This approach offers two main benefits. First, model designers are not required to learn new, possibly complex, extensions. Second, existing BPMN tools can be employed without the need for any

customization.

1.1.3 Bridging the Gap between Modeling and Implementation

Achieving a seamless transition from the modeling phase to the implementation phase has remained a persistently challenging task in MRS missions. This process typically involves transforming an abstract model into a concrete program ready for deployment on physical robots, requiring a thorough understating of both the high-level mission objectives and the low-level system intricacies.

This research is motivated by the aim to streamline this process, devising a systematic methodology for translating BPMN models into X-Klaim programs. This approach seeks to marry the best of the both worlds, combining the high-level abstraction and visual intuitiveness of the BPMN with the expressiveness and computational robustness of X-Klaim.

1.1.4 Contributions to the State-of-the-Art

In light of the challenges and motivations outlined, this research makes the following contributions to the state-of-the-art:

1. **Enhanced Programming and Coordination for Highly Heterogeneous MRSs:** Introduced a DSL tailored specifically for programming and coordinating highly heterogeneous MRSs, emphasizing its integration with a prominent robotic software framework (ROS).

2. **High-Level MRS Mission Prototyping:** Established a DSL focused on simplifying the prototyping of MRS missions, using selected elements to enhance mission visualization, catering especially to the intricacies of established robotic platforms.

3. **Streamlined Design-to-Implementation Process:** Presented a systematic methodology that ensures a seamless transition from high-level missions models to concrete programs, integrating established modeling notations with a robust programming paradigm to ensure efficient deployment in robotic environments.

1.2 Research Questions

The focus of this thesis is to explore how DSLs can be defined to program and coordinate highly heterogeneous multi-robot systems and model their missions while abstracting distribution and communication complexities. The

ultimate goal is to bridge the gap between high-level mission modeling and practical system implementation. To guide this investigation, we pose the following main research question and associated sub-questions:

Main Research Question:
RQ: "How can DSLs be defined to program and model highly heterogeneous multi-robots systems?"

Sub-Research Questions:
RQ1: "How can we define DSLs to facilitate efficient programming and high-level coordination in highly heterogeneous multi-robot systems?"

> This question delves into the intrinsic features of DSLs that support them in managing the complexities associated with programming and coordinating heterogeneous multi-robot systems. It also explores the potential of DSLs to simplify the communication process among multi-robot systems and enhance their coordination.

RQ2: "How can DSLs be used to model and abstract the mission of multi-robot systems?"

> This question focuses on identifying the unique attributes of DSLs that make them suitable for high-level mission modeling in multi-robot systems. It further delves into the ability of DSLs in simplifying complex mission characteristics and rendering them comprehensible for multi-robot systems.

RQ3: "How can the gap between high-level mission modeling and practical system implementation be bridged?"

> This question aims to explore the mechanism, processes, and strategies that can effectively mediate between high-level mission design and practical, real-word execution in multi-robot systems, thus facilitating seamless transition from conceptual mission modeling to on-ground deployment.

1.3 Structure and Contributions

This thesis is organized into four main parts, reflecting the progression of the research from foundational concepts to experimental evaluations and conclusions. Below an overview of the thesis' parts is provided, explaining for each chapter which research question is addressed, and which publications form the basis of the chapter. .

1.3.1 Part I - Introduction and Background

This part provides the foundational knowledge required for the rest of the thesis.

- **Chapter 2, Background:** An insight into crucial topics such as KLAIM and X-KLAIM languages, the principles of business process management, and the functionalities of BPMN 2.0. It particularly sheds light on the BPMN elements used in this study, and the property of well-structuredness of BPMN collaboration diagrams. Additionally, it offers a brief overview of MRSs, including their coordination and the use of ROS for robot software development.
- **Chapter 3, Systematic Literature Review on Domain-Specific Languages for ROS-based Systems:** A state-of-the-art review of languages used for multi-robot systems and ROS-based systems.

1.3.2 Part II - Coordinating and Programming MRSs

This part focuses on programming and coordination in multi-robot systems.

- **Chapter 4, Coordinating and Programming Multiple ROS-based Robots with X-KLAIM (Publications [20],[21],[22], RQ1):** presents how X-KLAIM enhances MRS coordination with intra- and inter-robot interactions in various scenarios, including simplified and enriched warehouse scenarios. The effectiveness and efficiency of the proposed approach are evaluated experimentally in terms of time and memory consumption.
- **Chapter 5, X-Klaim Mission Specification Patterns for ROS-Based Robots Systems (RQ1,RQ2):** elaborates the use of X-KLAIM to implement core movement patterns for ROS-based robots. These patterns, forming the basis for a variety of mission scenarios, demonstrate the flexibility and scalability of X-KLAIM in mission planning and execution.

1.3.3 Part III - From MRSs Models to Code

This part focuses on programming and coordination of multi-robot systems.

- **Chapter 6, Multi-Robot Mission Modeling using BPMN (Paper [23], RQ2):** introduces BPMN 2.0 collaboration diagrams as a tool for high-level modeling of cooperative behavior in MRSs, an approach that simplifies representation of complex interactions and coordination among robots, and a concrete application scenario illustrates its effectiveness.

- **Chapter 7, From BPMN to X-Klaim: A Systematic Methodology for Model Translation and Program Generation (Paper [24], RQ3):** bridges the gap between design and execution of MRSs, outlining a systematic method to translate BPMN diagrams into X-Klaim programs. The chapter elucidates the comprehensive mapping of BPMN elements to X-Klaim constructs, with a view to improving the efficiency and accessibility of MRS programming.

1.3.4 Part IV - Conclusions

Divided into two chapters.

- **Chapter 8, Concluding remarks:**, Summarises the work done in this thesis.
- **Chapter 9, Future work:**, presents areas and topics that could be investigated in the future.

Chapter 2

Background

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

2.1 KLAIM and X-KLAIM

2.1.1 KLAIM

KLAIM (Kernel Language for Agents Interaction and Mobility, [25]) is a formal language specially devised to design distributed applications consisting of possibly mobile software components deployed over the nodes of a network infrastructure. KLAIM is based on process calculi [26]. It generalizes the notion of *generative communication*, introduced by the coordination language Linda [27], to multiple distributed tuple spaces. A *tuple space* is a shared data repository consisting of a multiset of tuples. *Tuples* are *anonymous* sequences of data items that are associatively retrieved from tuple spaces using a *pattern-matching* mechanism. Communicating processes are decoupled 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. Interprocess communication occurs through the *asynchronous* exchange of tuples via tuple spaces: processes can indeed *insert*, *read*, and *withdraw* tuples into/from tuple spaces. Tuple spaces are identified through *localities*, which are symbolic addresses of network nodes where processes and tuples can be allocated. Localities themselves can be exchanged through interprocess communication.

A computational *node* of a KLAIM network is characterized by its locality and a collection of running processes. *Processes* are the active computational units of KLAIM and can be executed concurrently, either at the same locality or at different localities. Processes can execute basic actions acting on

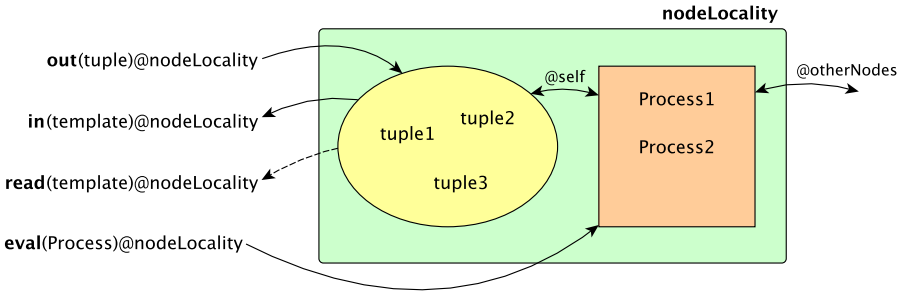


Figure 2.1: A KLAIM node.

network nodes, process variables, and process calls, either sequentially or in parallel. KLAIM supports *higher-order communication* since processes can exchange code and possibly execute it. Recursive behaviors are modeled via calls to process definitions.

Figure 2.1 depicts a generic KLAIM node and the basic *actions* which processes are made of. In these actions, processes can use the distinguished locality self to refer to their current hosting node.

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

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

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

2.1.2 KLAVA and X-KLAIM

The implementation of KLAIM consists of two main components:

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

KLAVA (KLAIM in Java, originally introduced in [28]) provides the implementation of the KLAIM concepts (Section 2.1) in terms of Java classes and methods, relying on the IMC framework [29] for the communication infrastructure. Any Java object can be stored into and retrieved from a KLAVA tuple, and the implemented pattern matching mechanism keeps Java subtyping into consideration. KLAVA strives to make Java programmers' life easier, but programmers still have to obey the rules of Java, particularly its verbosity. For this reason, we also developed X-KLAIM, a Domain-Specific Language (DSL) closer to KLAIM also providing typical high-level programming constructs. X-KLAIM (eXtended KLAIM) was initially introduced in [30] and reimplemented from scratch in [31]. 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 [31] is based on XTEXT [32], an Eclipse framework for developing programming languages and DSLs. XTEXT also provides complete IDE support based on Eclipse: editor with syntax highlighting, code completion, error reporting, and incremental building, to mention a few. Furthermore, we used another mechanism provided by XTEXT, that is, XBASE [33], an extensible and reusable expression language. By using XBASE in X-KLAIM, besides a rich Java-like syntax, we inherit its interoperability with Java and its type system. Thus, an X-KLAIM program can smoothly access any Java type and Java library available in the project's classpath. The interoperability with Java allowed us to integrate X-KLAIM seamlessly with the Java-ROS connector (see Section 4.1).

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

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

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

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

In particular, the name of a node also represents its locality within the network. Each node can specify some initialization code for creating and running a few processes, as shown in the examples of Section 4.2. This is the simplest way of specifying a *flat* network. X-KLAIM also implements the hierarchical version of the KLAIM model as presented in [34], but we will not use it in this paper.

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

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

The body consists of XBASE expressions, whose syntax has been extended with the KLAIM operations that we described in Section 2.1. 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, and it should be easily understood by Java programmers, but it removes much "syntactic noise" from Java. For example, terminating semicolons and other syntax elements like parenthesis when invoking a method without arguments are optional. Moreover, XBASE comes with a powerful type inference mechanism compliant with the Java type system: the programmer can thus avoid specifying types in declarations when they can be inferred from the context. Variable declarations start with `val` or `var` for final and non-final variables, respectively. The types of variables can be omitted if they can be inferred from the initialization expression.

In Figure 2.2, we show a simple code snippet of an X-KLAIM process body. The code should be easily readable by a Java programmer. We mention a few additional X-KLAIM syntax features to make the code more understandable. Such types as `String` and `Double` are Java types since, as mentioned above, X-KLAIM programs can refer directly to Java types. Similarly, `System.err.println` is the standard Java static method to print something on the screen. In most code snippets, we omit the Java-like import statements. Here we also see the typical KLAIM operations, `in` and `out`, acting on possibly distributed tuple spaces. Formal fields in a tuple are specified as variable declarations since formal fields implicitly declare variables that are available in the code occurring after `in` and `read` operations (just like in KLAIM). The X-KLAIM operation `eval` allows a process to start a new process concurrently at the specified locality. X-KLAIM provides syntactic sugar for collection literals: `#[ ... ]`. In the code snippet, `strings` is inferred to be of type `List<String>`. In X-KLAIM, *lambda expressions* have the shape `[ param1, param2, ... | body ]` where

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

Figure 2.2: An example of X-KLAIM code.

the types of parameters can be omitted if they can be inferred from the context. An X-KLAIM lambda expression can be used in any Java context where a lambda expression can be used, according to the Java type system: the type of the lambda expression must match the target Java functional interface. The code snippet shows standard Java stream operations using X-KLAIM lambda expressions.

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

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

2.2 BPMN

2.2.1 Business Process Management

Business Process Management (BPM) is a robust, comprehensive approach that focuses on improving the efficiency and effectiveness of an organization's operations. In its essence, BPM is concerned with designing, analyzing, executing, monitoring, and refining business processes [35], as depicted in Figure 2.3 [1]. This ongoing practice nurtures an environment of continuous improvement, with each iteration aimed at enhancing the quality of operations and the value delivered to stakeholders.

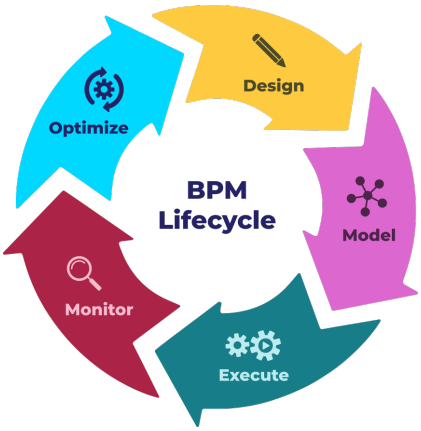


Figure 2.3: BPM Lifecycle

BPM is not a one-time project or a finite initiative. Rather, it is a perpetual practice that involves the constant analysis and refinement of business processes to meet evolving operational demands, organizational goals, and market dynamics. Its iterative cycle typically commences with process identification, proceeds with process discovery and analysis, progresses towards process redesign and implementation, and continues with consistent monitoring and refinement.

BPM is a critical bridge connecting business strategy and operational implementation. By offering a lucid visualization of processes, it allows stakeholders to gain an in-depth understanding of the complex interdependencies between different tasks, identify potential bottlenecks, and make data-driven decisions regarding process enhancement and resource allocation. The cyclic nature of BPM ensures that processes remain relevant, adaptable and consistently optimized in response to changing business landscapes.

[1] Figure source: `https://www.yworks.com/assets/images/use-cases/bpm/bpm-illustration.cf19f42ad5.svg`

2.2.2 Business Process Model and Notation 2.0

Introduced by Object Management Group (OMG), Business Process Model and Notation 2.0 (BPMN 2.0) is an essential standard for visualizing and managing business processes [36]. Its shared language bridges the divide between business stakeholders and IT professionals, fostering a mutual understanding of complex business operations [37].

At its core, BPMN 2.0 stands as a graphical tool that defines the steps and order of a business process. It encompasses a set of symbols and conventions that allow businesses to create process diagrams, outlining the sequence of activities that lead to a certain outcome. This, in turn, provides a clear, visual roadmap for better understanding, optimization, and automation [38, 39, 40].

Yet, BPMN 2.0 moves beyond simple process flow diagrams. Its strength lies in its ability to visualize intricate collaboration scenarios [23, 18, 41, 42]. By using collaboration diagrams, BPMN 2.0 enables organizations to represent interactions among different participants or processes, whether they are within the same business entity or across multiple organizations. This visual representation fosters a shared understanding among stakeholders, promoting effective collaboration and reducing miscommunications [42, 41].

BPMN 2.0 also stands out with its expressive language for modeling communication and coordination mechanisms [43]. By leveraging a variety of events, messages, and gateways, BPMN 2.0 effectively depicts the communication flow, synchronization points, and conditional paths within and between processes [44].

In the following section, we will delve into the specifics of the BPMN 2.0 notation. This will provide a more comprehensive understanding of how this versatile tool achieves its powerful visual representation of business processes ultimately fostering efficient communication, process management, and collaborative efforts.

2.2.3 BPMN Notation

The scope of BPMN 2.0 extends to a vast array of over 85 elements, each serving specific roles within the process modeling environment. However, not every element is equally critical for all applications. For the purpose of this thesis, a subset of these elements has been carefully chosen to facilitate the efficient depiction of processes, collaborations, and coordination mechanisms. The selection was guided by a top-down approach driven by modeling activity across different scenarios and a bottom-up approach inspired by experiments with various implementations.

Here is a brief overview of the BPMN 2.0 elements selected for use in this thesis:

- **Collaboration Diagrams**: These are visual representation of multiple interacting processes. Each diagram contains pools which can be multi-instance housing various processes.

- **Pools** (Figure 2.4): are used to represent participants or organisations involved in the collaboration, and include details on internal process specifications and related elements. Pools are drawn as rectangles, and they usually have a name associated with, referring to the name of the organisation. BPMN allows to assign a multi-instance marker (three vertical lines) to a pool, representing multiple instances playing the same role.

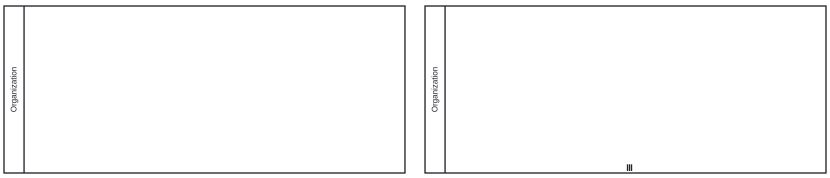


Figure 2.4: BPMN Pools.

- **Processes**: These are sequences of activities, gateways, and events that are interconnected by sequence flows. Data objects are also associated with these elements, depicting the flow of information and material.

- **Activities** (Figure 2.5): These represent the work to be performed within a process. Notably, a *Call Activity* calls another process, allowing for the structuring of large and complex models as decoupled reusable processes. A *Script Task* signifies the execution of a piece of code.

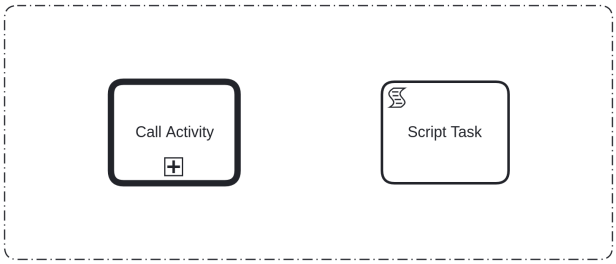


Figure 2.5: Considered BPMN Activities.

- **Event Sub-Process** (Figure 2.6): This is a sub-process that is not part of the normal flow of its parent process.

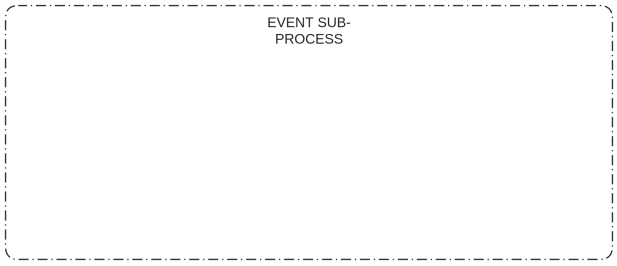


Figure 2.6: BPMN Event Sub-Process.

- **Connecting Edges** (Figure 2.7) are used to connect process elements inside or across different pools. Sequence Edges are solid connectors used to specify the internal flow of the process, thus ordering elements in the same pool, while Message Edges are dashed connectors used to visualize communication flows between organizations.

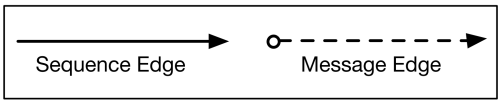


Figure 2.7: BPMN Connecting Edges.

- **Gateways** (Figure 2.8): Gateways offer a sophisticated mechanism to govern the flow of a process, enabling the representation of parallel activities and decision-making paths. This is achieved through four types of gateways: *Exclusive (XOR)*, *Parallel (AND)*, *Inclusive (OR)*, and *Event-Based*. *Exclusive* gateways, denoted " ˆ ", facilitate singular decisions in the process flow, activating only one outgoing path. *Parallel* gateways, denoted by " + ", enable the simultaneous execution and synchronization of multiple process branches. *Inclusive* gateways, marked with an " ◯ ", introduce flexibility by allowing any number of non-mutually exclusive outgoing paths. *Event-based* gateways, depicted with a double-rounded " ⬡ ", control flow based on the occurrence of external events. These gateways, equipped with guard conditions on their outgoing sequence edges, provide an expressive and nuanced toolset for modeling complex decision-making processes and parallel execution paths.

- **Events** (Figure 2.9): These serve as vital components, capturing the diverse dynamics that occur at the initiation, during the unfolding, and at the culmination of a process. Notably, *Timer* and *Conditional events* act in response to time-bound or condition-specific changes, adding flexibility and adaptability to the model. Equally important, *Error events* address unexpected anomalies during execution, contributing to

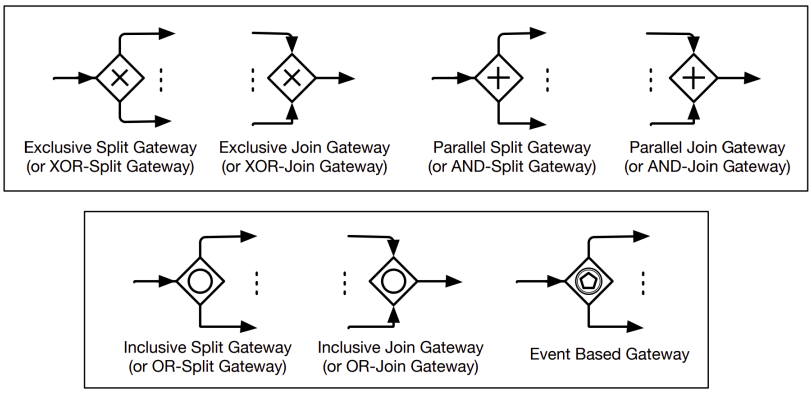


Figure 2.8: Considered BPMN Gateways.

the robustness of the process model. On the other hand, communication within and between processes is elegantly handled by *Signal* and *Message* events. *Signal events* embody broadcast communications akin to a multicast mechanism, while *Message events* represent directed, unicast exchanges between specific entities. To signify the abrupt end of all activities within a process, *Terminate* events are used. They serve as definitive markers for the conclusion of all ongoing processes within a pool. Lastly, there are *End events*. These are employed to denote the point where a process concludes under normal circumstances, providing a clean and clear endpoint to a process flow.

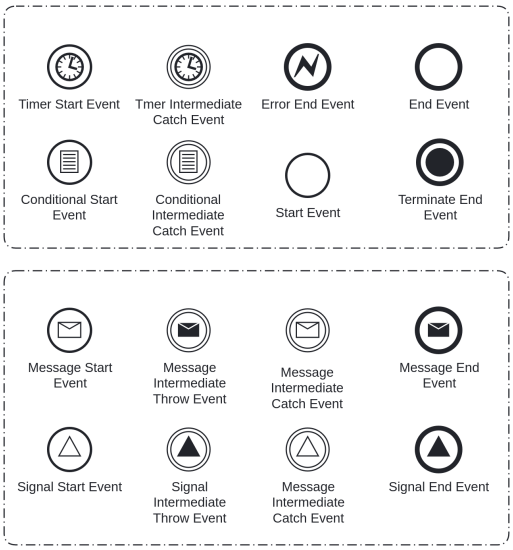


Figure 2.9: Considered BPMN Events.

Data Objects (Figure 2.10): represent information and material flowing in and out of activities. They are depicted as a document with the

upper-right corner folded over, and linked to activities with a dotted arrow with an open arrowhead (called *data association* in BPMN).

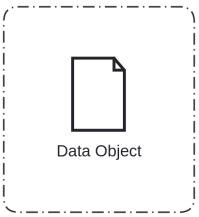


Figure 2.10: Considered BPMN Data Object.

2.2.4 Well-structuredness in BPMN collaborations

Understanding the concept of well-structuredness in BPMN is essential, given its pivotal role in maintaining coherence and integrity in process modeling and collaboration. Well-structuredness is a property that emphasizes the organization and flow of a business process, adhering to certain structural rules to ensure process consistency and clarity [45]. This property becomes particularly crucial when translating a BPMN collaboration into another language paradigm, such as X-KLAIM, which we focus in this thesis.

This principle implies that every node with multiple outgoing edge (a 'split') should have a corresponding node with multiple incoming edges (a 'join'). This pattern forms a structured core, providing a Single Entry Single Exit (SESE) process component, often perceived as a well-structured unit [46].

The structured core can encompass various process elements including activities, intermediate events, or composite processes initiated with (AND, XOR, Event-based) and concluded with a corresponding join. Loops are also permitted within a well-structured model; however, they must be formed through XOR gateways to maintain the well-structuredness.

When this property is applied to a BPMN collaborations, it necessitates that all processes involved in a collaboration are well-structured. This promotes effective communication and collaboration by ensuring each process is understood in the same way by all participants [47]. In essence, well-structuredness is a fundamental attribute in BPMN, greatly influencing process efficiency, understandability, and manageability.

2.3 Robotics

Robotics is a scientific and engineering discipline that is focused on the understanding and use of artificial, semi-autonomous systems that can interact

physically and socially with the environment, and perform tasks with various degrees of autonomy [48].

2.3.1 History of Multi-Robot Systems (MRS)

The concept of MRS, despite being relatively recent in the annals of robotic research, has already left an indelible mark on the field, from both practical and theoretical perspectives. It is a discipline at the intersection of robotics, computer science, and control theory that seeks to understand and design systems of multiple robots working in concert [49].

The genesis of MRS can be tracked back to the late 1980s and early 1990s when researchers started envisioning the benefits of multiple robots working collectively to perform tasks. This was a period marked by the pioneering works of researchers like Hiroaki Yamaguchi and Lynne E. Parker who demonstrated the potential of cooperative robots [50, 51].

In the 2000s, the research in MRS took a leap forward with the introduction of bio-inspired algorithms, drawing parallels from nature, like ant and colonies and bird flocks. This led to the development of swarm robotics, a subfield of MRS where large numbers of relatively simple robots cooperate to perform tasks [52].

At the same time, the focus shifted towards problems related to communication, sensing and processing in MRS. Researchers started working on communication protocols, sensing techniques, and distributed algorithms to handle the complexities of MRS [53].

The advent of Robotic Operating System (ROS) in 2007 further bolstered the development of MRS by providing a robust and versatile platform for designing complex robot systems [6] (Section 2.3.3). The focus now includes not just homogeneous systems (Figure 2.11) where all robots are identical but also heterogeneous systems (Figure 2.12) where robots have different capabilities and roles.

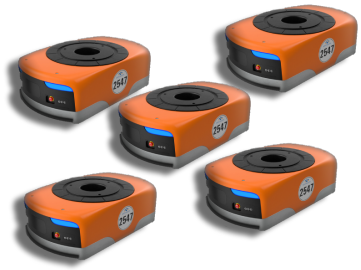


Figure 2.11: Homogeneous MRS.

Figure 2.12: Heterogeneous MRS.

Today, MRS has found a wide range of applications, from environmental monitoring [54, 55], search and rescue operations [56], to space exploration [57] and more. Despite its rapid growth, MRS still poses many fascinating and challenging problems that continue to attract researchers worldwide [58, 59].

2.3.2 Coordination in MRS

Coordination in MRS is a sophisticated and intricate area of study, embodying the interactions, collaborations, and shared task achievement of multiple robots. Understanding its dynamics not only requires a grasp of signal robot operations but also an intricate perception of how multiple autonomous agents can work in unison while navigating complex environments and performing various tasks [58, 49, 60, 61, 62].

The idea of coordination in MRS is inspired by biological systems such as flocks of birds [63], colonies of ants [64, 65], and schools of fish [66], where simple local rules lead to complex and coherent global behaviors [59]. The goal in MRS coordination is to engineer these complex behaviors in a more controlled, reliable, and predictable manner.

A novel taxonomy was proposed to classify the various approaches to coordination within MRS [61, 60]. This taxonomy is structured around two main groups of dimensions: the coordination dimensions and the system dimensions. These dimensions represent distinct features that have been collectively organized within the taxonomy.

The coordination dimensions encompass the areas of cooperation, knowledge, coordination, and organization. Meanwhile, the system dimensions include communication, team composition, system architecture, and team size (Table 2.1).

Coordination Dimensions	**System Dimensions**
Cooperation	Communication
Knowledge	Team composition
Coordination	System architecture
Organization	Team size

Table 2.1: Taxonomy Classification Dimensions.

This taxonomy provides a hierarchical structure for the coordination dimensions. These include the cooperation level, the knowledge level, the coordination level, and the organisation level (Figure 2.13).

This first level of the taxonomy, the cooperation level, is related to system's capacity to collaborate for the accomplishments of specific tasks. The

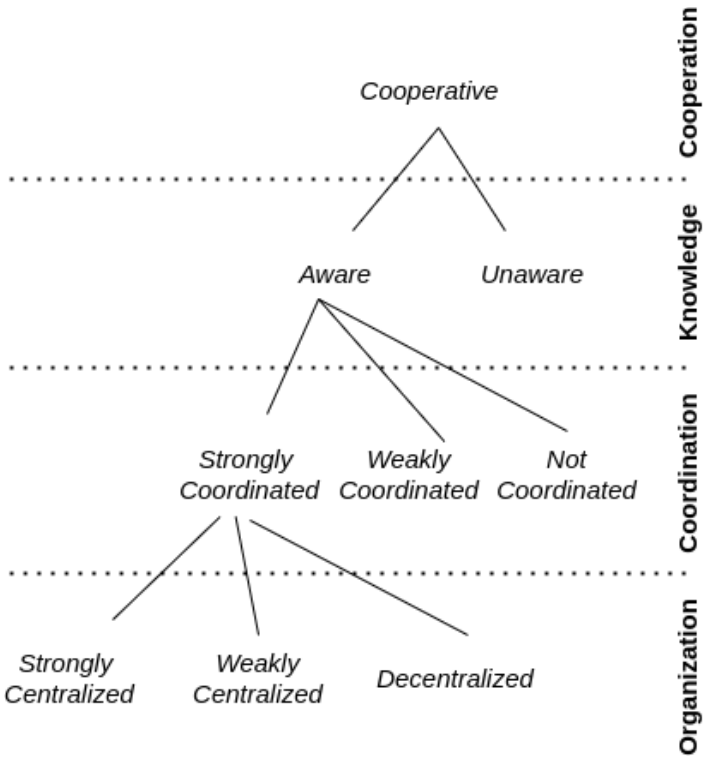


Figure 2.13: Hierarchical structure of the MRS coordination.

second level, the knowledge level, pertains to the extent of awareness each robot has about the presence of other robots within the system. The third level, the coordination level, concerns the mechanism employed to achieve system-wide cooperation. The fourth and final level, the organization level, considers how the decision-making is implemented within the MRS. Here, two predominant strategies emerge: *centralized coordination* and *decentralized coordination* [67, 68].

In *centralized coordination*, a designated entity or 'leader' presides over the actions of all the robots in the system. This can be strongly centralized where leadership is constant, or weakly centralized, allowing dynamic changes in leadership roles throughout the operation [69].

Conversely, *Decentralized coordination* empowers each robot to make autonomous decisions based on locally available information, without a specific leader. Here, coordination is achieved through peer-to-peer interactions and mutual adjustments of actions [50].

2.3.3 ROS

Robotic Operating System (ROS)[2] is one of the most sophisticated and popular frameworks for writing robot software. ROS has its origins in the late 2000s, fundamentally altering the way researchers and developers approach multi-robot systems and robotic applications as a whole [6].

In 2007, Willow Garage, a California-based robotics research lab, began the development of ROS. Their objectives was to provide a unified and flexible framework for writing robust robot software aiming to mitigate the challenge of writing custom code for each robotic application [70].

ROS provides tools and libraries for simplifying the development of complex controllers while abstracting from the underlying hardware. ROS works with more than a hundred types of robots, ranging from autonomous cars to drones and humanoid robots, and integrates many sensors.

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

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

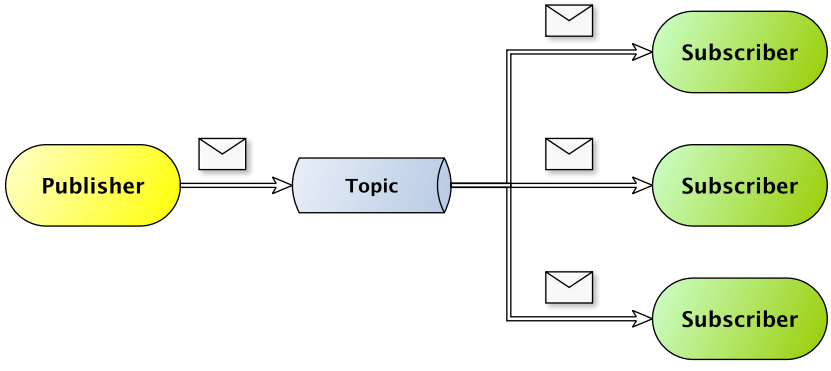


Figure 2.14: ROS publish/subscribe mechanism.

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

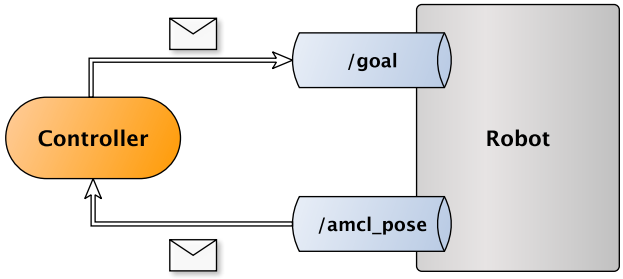


Figure 2.15: Interactions within a mobile ROS robot.

Multiple publishers and subscribers for the same topic are allowed. The diagram in Figure 2.14 illustrates this concept. In contrast, the one in Figure 2.15 shows how a robot controller interacts with the devices of a mobile robot in a black-box, hardware-independent fashion. In the latter diagram, the controller acts as both publisher and subscriber. As a publisher, it sends a message directed to the navigation node responsible for actuating the wheels. At the same time, as a subscriber, it receives back messages containing the robot's actual position. The topic `/goal` stands for the *goal* position that the mobile robot should attempt to reach. The topic `/amcl_pose` stands for the estimated *pose* of the robot, in a known map, calculated from the robot sensors data with the "Adaptive Monte Carlo Localization" approach.

Chapter 3

Systematic Literature Review on Domain-specific Languages for ROS-based Systems

Recent years have seen a surge in advancements in the field of robotics, driven by the growing demand for intelligent, autonomous systems that can execute complex tasks across various domains. Designing and implementing effective software systems poses a significant challenge when developing robotic applications. This process can be streamlined using domain-specific languages, offering custom solutions to meet the unique requirements of robotics. These languages aid developers in modeling and programming individual robots, and in coordinating multi-robot systems more efficiently.

Inspired by [3], this systematic literature review (SLR) sets out to examine the current state-of-the-art of domain-specific languages for robotics and ROS-based systems. The focus will be on how these languages tackle challenges in modeling, programming, and coordinating (multi-)robot systems. Various sub-domains of robotics will be explored, including task and behavior specification, robot coordination and collaboration, perception and sensing, manipulation and grasping, robot modeling and simulation, human-robot interaction (HRI), robot programming languages, robotic architecture, and safety & security.

In this chapter, a comprehensive overview of pertinent literature is presented, discussing the various domain-specific languages and their applications in the field of robotics. These languages will be categorized based on their features, capabilities, and targeted application domains. The objective is to improve understanding of the existing landscape and identify potential areas for future research. This systematic literature review will clearly outline our contributions and demonstrate how they advance the current state of the art.

3.1 SLR Methodology

3.1.1 SLR Questions

The systematic literature review process is guided by research questions that focus on analyzing the current state of domain-specific languages in robotics. The primary research question aims to cover the overall state of the art, while the secondary research questions explore specific facets of the subject.

Primary Question

SLR-Q: What are the key features, capabilities, and applications of Domain-Specific Languages in the field of robotics, particularly in the context of multi-robot systems and ROS-based systems?

Secondary Questions

SLR-Q1: Which functional aspects are typically addressed with DSLs in robotics?

SLR-Q2: What is the distribution of DSL types used in the field of robotics?

SLR-Q3: Do DSLs support multi-robot systems and heterogeneous robots?

SLR-Q4: Do DSLs address coordination and decentralized coordination among robots?

SLR-Q5: What types of programming languages are typically generated by DSLs for robotics?

SLR-Q6: Do DSLs integrate with Integrated Development Environments (IDEs)?

SLR-Q7: What is the prevalence of formal languages in DSLs for robotics?

3.1.2 Search Strategy

A systematic search strategy was utilized to identify relevant articles for our research. This strategy involved multiple databases and search engines, including Scopus, IEEE Xplore, ACM Digital Library, ScienceDirect, and Google Scholar. The search focused on key terms and phrases relating to robotics, domain-specific languages, ROS-based systems, coordination, and multi-robot systems.

The search terms were refined as follows:

- Domain-specific languages and modeling notations: "domain-specific language", "domain-specific modeling language", "formal language", "specification language", "description language", "code generation", "DSL", "DSML", "meta-modeling", "metamodel", "model-driven engineering", "model-driven software engineering", "model-driven architecture", "MDE", "MDSE", "MDD".

- Robotics and multi-robot systems: "robot", "ROS", "coordination", "multi-robot", "swarm", "distributed systems", "coordination language", "coordination model", "multi-agent", "multi-agent system", "MAS", "cooperative".

Using these terms in conjunction with appropriate filters, we ensured a comprehensive literature review covering the overlap of domain-specific languages and multi-robot systems.

3.1.3 Inclusion and Exclusion Criteria

To direct the systematic literature review towards relevant studies, we established clear inclusion and exclusion criteria. These rules aid in selecting articles that directly address the research questions and contribute to the understanding of domain-specific languages in robotics.

Inclusion criteria

IC1: Publication was published in one of the specified conferences, workshops, or journals related to robotics or model-driven engineering.

IC2: Publication includes keywords related to domain-specific languages, metamodeling, robotics, and coordination.

Exclusion criteria

EC1: The paper does not focus on domain-specific languages, modeling languages, or related concepts.

EC2: The paper does not discuss robotics or related concepts.

EC3: The paper published prior to 2012 is not considered, as the scope of the study encompasses the ten-year period preceding the conclusion of the SLR at the end of 2022.

EC4: The paper is not available in a repository or does not have open access.

3.1.4 Search and Filtering Process

Our systematic search strategy, as detailed in Section 3.1.2, was applied across multiple databases and search engines as previously mentioned. After the initial search was performed, we used advanced search features and filters to refine the results. We filtered for papers published within a ten-year period, from 2012 to 2022. We also applied filters to limit results to specific conferences, workshops, and journals as listed in Table 3.1, adhering to our first inclusion criterion (IC1).

After applying these search criteria and filters, we refined the initial collection of 2,103 papers down to 211. These shortlisted papers were then subjected to the screening process that led to the final selection of 29 papers, which were carefully analyzed and used for data extraction.

Table 3.1: Considered Conferences, Journals, and Workshops in Robotic and Computer Science

Robotic Field	**Computer Science Field**
IEEE Intl. Conf. on Robotic Computing (IRC), Wiley Journal of Field Robotics (JFR), IEEE Robotics and Automation Letters, Robotics: Science and Systems Conference (RSS), Intl. Journal of Advanced Robotic Systems, Robotics and Autonomous Systems, Springer Proceedings in Advanced Robotics, IEEE Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR), Intl. Conf. on Control, Automation, Robotics and Vision (ICARCV), ACM/IEEE Intl. Conf. on Human-Robot Interaction, Autonomous Agents and Multi-Agent Systems, Intl. Conf. on Control, Automation and Systems, Intl. Journal of Robotics Research, Science Robotics, Intl. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE International Conference on Robotics and Automation (ICRA)	ACM Transactions on Autonomous and Adaptive Systems, IEEE/ACM International Conference on Software Engineering (ICSE), IEEE Transactions on Software Engineering, Information and Computation, Intl. Journal on Software Tools for Technology Transfer, Journal of Logical and Algebraic Methods in Programming, Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS), Proceedings Intl. Computer Software and Applications Conference, Computer Languages Systems and Structures, Computer Science and Information Systems, Conference on Model-driven Engineering and Software Development (MODELSWARD), International Conference on Coordination Models and Languages, International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)

3.1.5 Screening and Selection Process

The pool of 211 papers, derived from the search and filtering process, underwent a preliminary screening. This screening involved a review of each paper's abstract, conclusions, and keywords to ascertain their relevance. Papers failing to meet the inclusion and exclusion criteria were removed from further consideration.

Following this screening, we chose 29 papers for a full-text review. This step involved a thorough reading and analysis of each paper, during which we extracted relevant information. We then synthesized this data in line with the categories detailed in Section 3.1.6. For more in-depth look at this analysis, refer to Appendix A, which houses a comprehensive classification table.

3.1.6 Data Extraction and Synthesis

In our systematic literature review, we extracted data from the 29 selected papers and classified them according to the following categories:

Sub-domain We classified the DSLs based on the specific sub-domain within robotics, an approach inspired by [3]. The identified sub-domains comprise:

- **Human-robot interaction** Focused on facilitating communication and collaboration between humans and robots.
- **Robot coordination and collaboration** Concerned with the management of multiple robots working together towards a common goal.
- **Manipulation and grasping** Addresses the challenges related to robot manipulation and grasping of objects in their environment.
- **Robot programming** Aims to simplify the programming of robot behavior and control logic.
- **Robotic architecture** Deals with the design and implementation of the software architecture for robot systems.
- **Security and safety** Focuses on ensuring the safe and secure operation of robots and their interactions with the environment and humans.
- **Task and behavior specification** Involves defining and describing the tasks and behaviors that a robot should perform.
- **Robot modeling and simulation** Provides tools and techniques for modeling and simulating robot systems for design, analysis, and testing purposes.

Formal Language DSLs that provide formal semantics, syntax, or a well-defined mathematical model for their constructs.

Multi-robot Support DSLs that explicitly support multi-robot systems or provide abstractions for these types of systems.

Heterogeneous Robot Support DSLs that cater to the needs of heterogeneous robot systems, involving robots with different characteristics and capabilities.

Coordination Support DSLs that provide abstractions for robot coordination or address coordination in an explicit way.

Decentralized Coordination Support DSLs that provide abstractions or mechanisms for decentralized coordination among robots.

IDE Integration DSLs that offer integration with a development environment to facilitate editing, debugging, and testing.

3.2 Results and Analysis

3.2.1 Sub-domain Distribution in Robotics DSLs (RQ1)

Our analysis of the sub-domains targeted by the reviewed DSLs reveals a diverse range of application areas within robotics. As depicted in Figure 3.1, the most prevalent sub-domains include robot programming (48.28%), robotic architecture (44.48%), and security & safety (41.38%). These results suggest that the design and development of DSLs are primarily focused on improving the programmability, architectural design, and safety aspects of robotic systems. Other sub-domains such as robot coordination and collaboration (31.03%), task and behavior specification (17.24%), and robot modeling and simulation (13.79%) also receive attention, highlighting the demand for DSLs that cater to these specific needs. However, areas such as human-robot interaction (6.90%) and perception and sensing (3.45%) appear to be less explored, indicating potential opportunities for future research and development in these sub-domains.

The work of this thesis focuses on the robot programming, robot coordination and collaboration, and task and behavior specification.

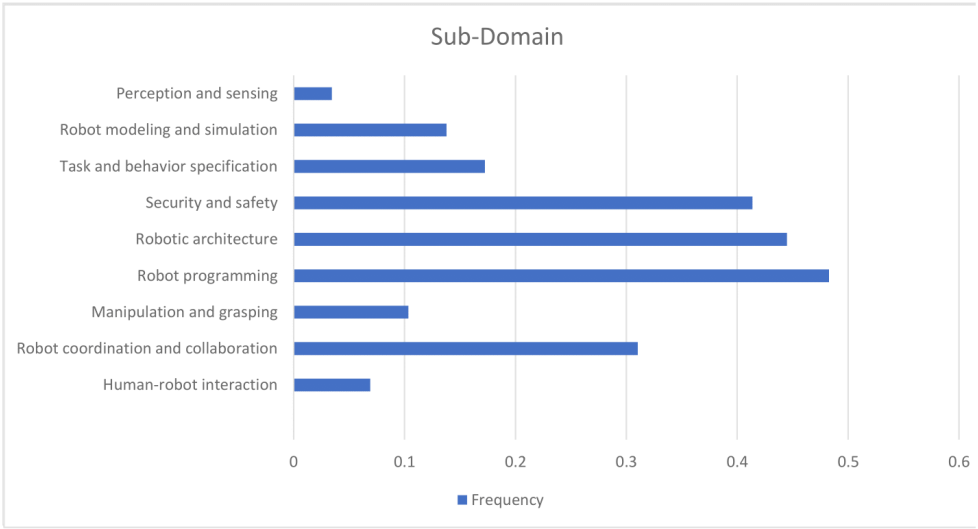


Figure 3.1: Distribution of subdomains in DSLs for robotics

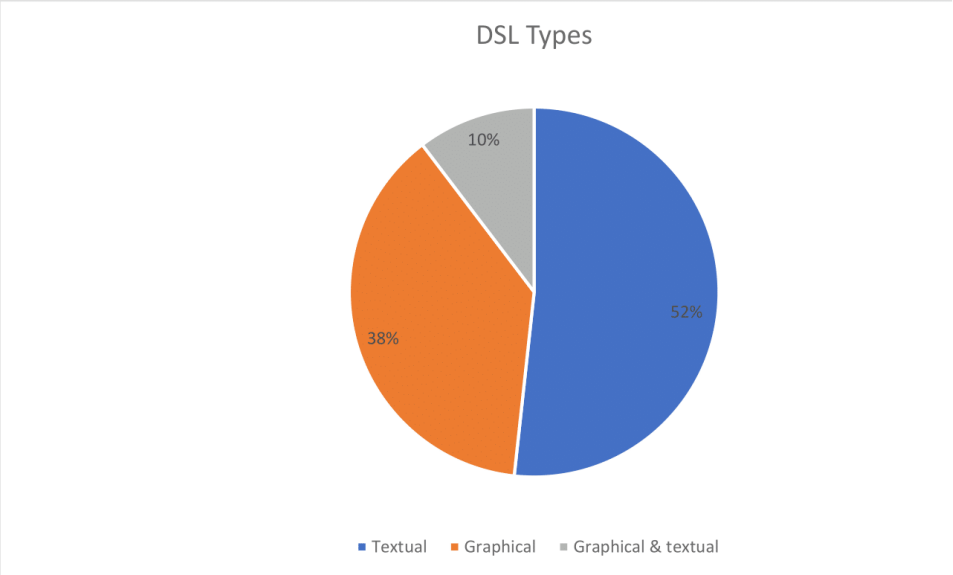


Figure 3.2: DSL type distribution in robotics

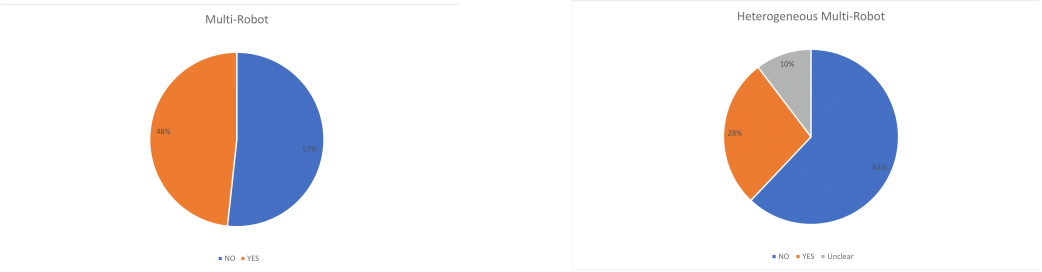
3.2.2 DSL Type Distribution in Robotics (RQ2)

Our analysis of the DSL types employed in the reviewed robotics literature reveals a different types of languages. As depicted in Figure 3.2, the most common DSL type seems to be the textual one, with 15 instances found in our review. This prevalence might be attributed to the familiarity and ease of the use associated with textual representations, which resemble traditional programming languages. Graphical DSLs account for 11 instances in our review, reflecting a growing interest in visual representations that can intuitively depict complex robotic systems and interactions. This approach proved especially beneficial for those with minimal programming experience, as it facilitates their active involvement in the process. As users gain familiarity with robotics applications, the textual notation grows increasingly useful for managing larger-scale applications with more specific requirements. This enhances their ability to interact effectively with the process. Additionally, 3 instances of DSLs combine both graphical and textual elements, providing users with the flexibility to choose the most suitable representation based on their needs and preferences. This hybrid approach may enhance the usability and expressiveness of the DSLs, catering to a wider range of users and applications within the robotics domain.

The work of this thesis provides both textual and graphical DSLs.

3.2.3 DSL Support for Multi-robot Systems and Heterogeneous Robots (RQ3)

In response to RQ3, we found that 14 of the reviewed DSLs explicitly support multi-robot systems or provide abstraction of such systems, showcasing a growing interest in designing languages that cater to the needs of multi-robot systems. Moreover, our review identified 8 DSLs that support heterogeneous robot systems, which consist of robots with diverse characteristics and capabilities.



(a) Supports Multi-robot (b) Supports Heterogeneous Robots

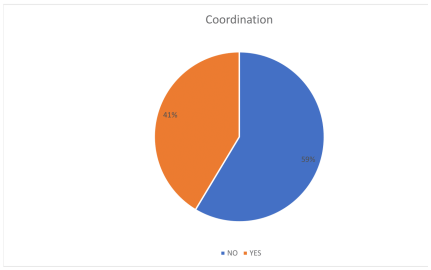
Figure 3.3: Distribution of DSLs supporting multi-robot systems and heterogeneous robots

However, we noticed a significant gap in DSLs addressing distribution and high heterogeneity in multi-robot systems. In fact, only few DSLs stands alone in addressing distribution among the DSLs studied [71, 72, 73], which is crucial aspect of effective resource allocation, load balancing, and task assignment in multi-robot systems. Furthermore, high heterogeneity, which promotes versatility and adaptability in operations, is also addressed only by a small subset of DSLs [71, 72, 73]

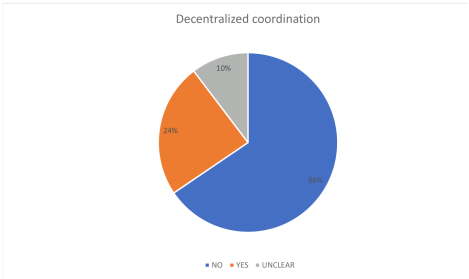
The work of this thesis supports multi-robot systems, addressing distribution and high-heterogeneity.

3.2.4 DSL Support for Coordination and Decentralized Coordination in Multi-robot Systems (RQ4)

In response to RQ4, we analyzed the reviewed DSLs concerning their support for coordination and decentralized coordination in multi-robot systems, whether or not they include build-in constructs or libraries that facilitate the coordination of multiple robots, such as synchronization, communication, or task allocation. As depicted in Figure 3.4a, almost half of the reviewed DSLs support the coordination which indicates an increasing focus on facilitating collaboration among multiple robots.



(a) Supports Coordination



(b) Supports Decentralized Coordination

Figure 3.4: Distribution of DSLs supporting coordination and decentralized coordination in multi-robot systems

Yet, only 7 DSLs explicitly support decentralized coordination, suggesting that there is room for improvement in the development of languages that cater to distributed control and decision-making in multi-robot systems. Decentralized coordination is crucial in scenarios where centralized control is either not feasible or undesirable due to communication limitations, dynamic environments, or the need for increased robustness and scalability. We also observed that only a few DSLs provide high-level abstractions for coordination, which could simplify the design, implementation, and verification of coordination strategies in multi-robot systems. Addressing this gap may result in more accessible and efficient tools for designing and programming coordinated multi-robot systems, ultimately enhancing their performance and adaptability in complex, real world scenarios. The limited number of DSLs supporting coordination, decentralized coordination of multi-robot systems at a high-level is noticeable. This highlights potential areas for research and development in domain-specific languages.

The work presented in this thesis primarily focuses on high-level coordination within a decentralized topology of multi-robot systems.

3.2.5 Code Generation in DSLs for robotics (RQ5):

In response to RQ5, we examined the code generation capabilities of the reviewed DSLs for robotics, as shown in Figure 3.5. The ability to generate code in widely used programming languages is essential for ensuring compatibility and ease of integration with existing software and hardware platforms.

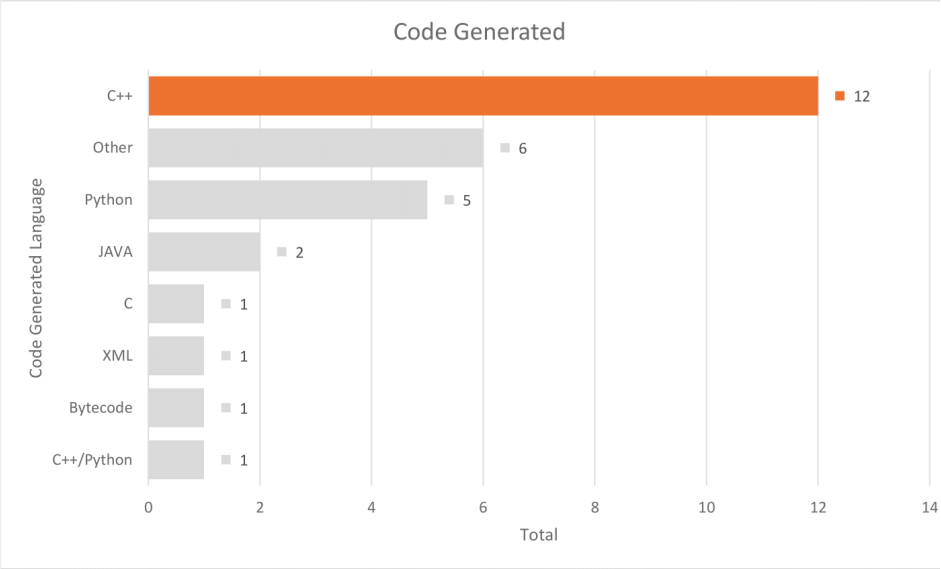


Figure 3.5: Distribution of code generation languages in DSLs for robotics

Out of the 29 DSLs analyzed, C++ is the most common target language for code generation, with 12 DSLs generating C++ code. This is likely due to the widespread adoption of C++ in the robotics community, thanks to its performance benefits and mature performance. Python is the next most common target language, with 5 DSLs generating Python code, reflecting its popularity as a versatile and user-friendly language. Other code generation languages include Java (2 DSLs), C(1 DSL), XML (1 DSL), Custom Bytecode(1 DSL), and a combinaison of C++/Python (1 DSL). Additionally, 6 DSLs generate code in other languages not listed here.

The target language in the thesis work is Java.

3.2.6 IDE in DSLs for robotics (RQ6):

In the analysis of the 29 DSLs, we found that 13 instances of DSLs for robotics explicitly offered integration with Integrated Development Environment (IDEs), as shown in Figure 3.6. This integration plays vital role in streamlining the development process and enhancing the user experience. By incorporating IDE support, DSLs for robotics can significantly improve productivity, provide debugging and testing capabilities, and facilitate better collaboration among development teams. Furthermore, IDE integration often allows for more accessible entry points for users who are new to robotics or DSLs, promoting broader engagement in the field.

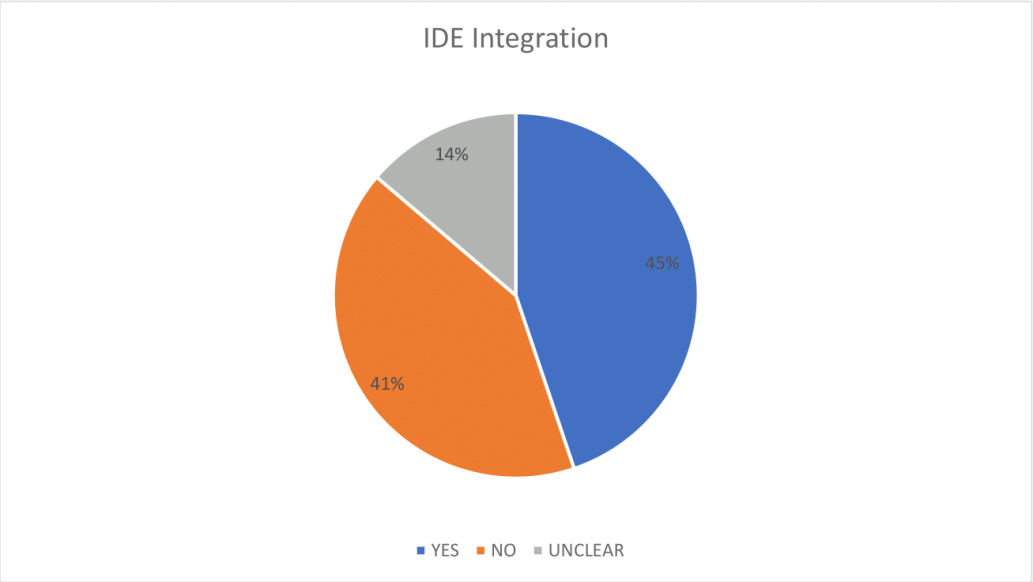


Figure 3.6: Distribution of IDE integration in DSLs for robotics

Overall this trend reflects the growing importance of comprehensive tooling and environments for successful development and deployment of robotics applications.

The DSL considered in this thesis offers an integration with Eclipse.

3.2.7 Prevalence of formal languages in DSLs for robotics (RQ7):

The analysis of the 29 selected papers reveals that 13 of them (44.8%) employ formal languages in their DSLs for robotics, as depicted in Figure 3.7. Formal languages contribute to system analysis and verification by providing well-defined syntax, semantics, and mathematical models for their constructs. This ensures a higher level of precision, consistency, and correctness in the design and implementation of robotic systems. The use of formal languages allows for rigorous analysis of robotic systems, which can help identify potential errors, inconsistencies, or vulnerabilities before deployment. Additionally, formal languages enable the verification of system properties, ensuring that the implemented robotic system meet their design specification and requirements.

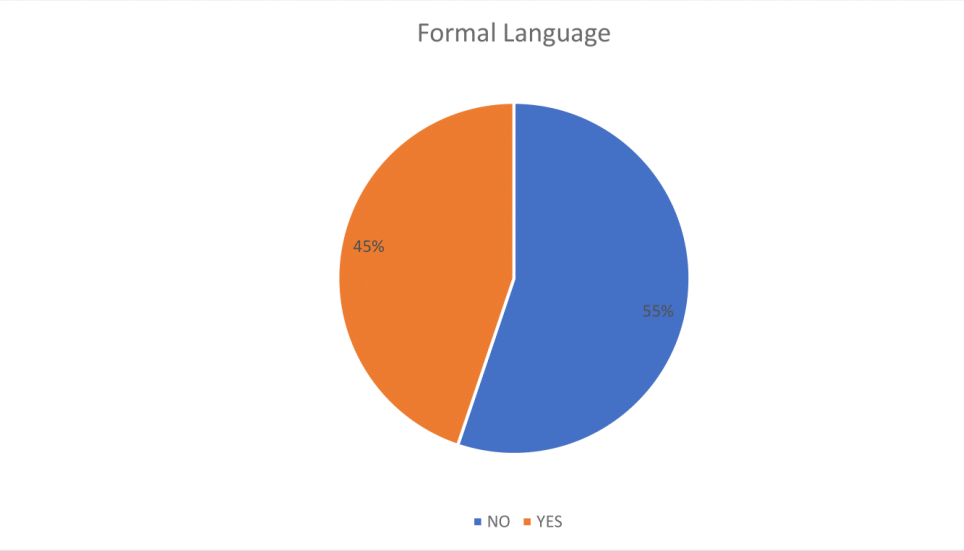


Figure 3.7: Distribution of formal DSLs for robotics

The prevalence of formal languages in DLSs for robotics highlights the importance of rigorous analysis and verification techniques in the development of complex robotic systems, ultimately contributing to the safety, reliability, and robustness of these systems.

The work of this thesis is based on a formal language.

PART II

COORDINATING AND PROGRAMMING MRSS

Chapter 4

Coordinating and Programming Multiple ROS-based Robot with X-Klaim

The complexity of software development for robotics applications increases significantly when dealing with Multi-Robot Systems. To address these challenges, this study proposes an approach for programming MRSs at a high abstraction level using X-Klaim. X-Klaim's computation and communication of both intra- and inter-robot interactions, allowing developers to focus on MRS behavior and achieve readable, reusable, and maintainable code.

4.1 The X-Klaim approach to multi-robot programming

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

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

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

In X-Klaim, the distributed architecture of the MRS's software is naturally rendered as a network where the different parts are deployed. As shown in Figure 4.1, we associate an X-Klaim node to each robot of the MRS.

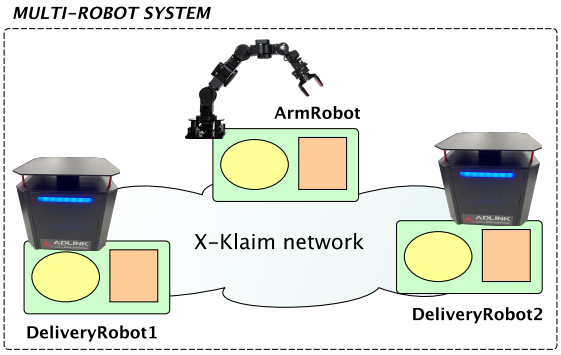


Figure 4.1: Software architecture of an MRS in X-KLAIM.

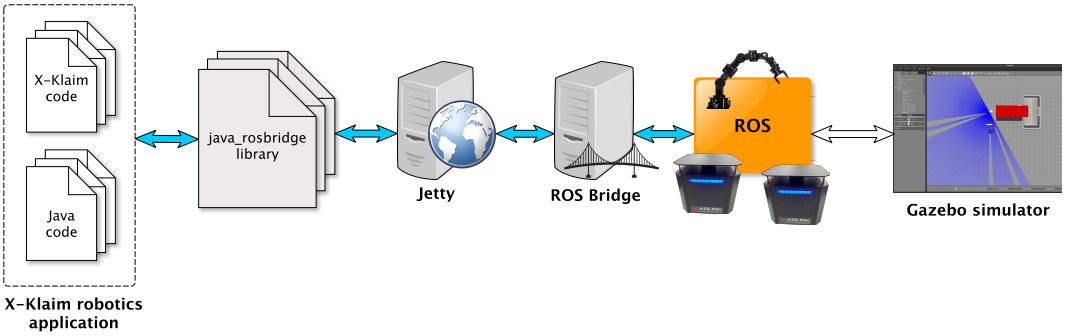


Figure 4.2: The integrated framework.

In its turn, the internal distribution of the software controller of each robot is managed by concurrent processes that synchronize their activities using tuples stored in the robot's tuple space. Inter-robot interactions rely on the same communication mechanism by specifying remote tuple spaces as targets of communication actions.

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

In practice, to program the behaviors of the robots forming an MRS, we

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

Figure 4.3: Example of a JSON message for the `/goal` topic.

enabled X-KLAIM programs to interact with robots' physical components by integrating the X-KLAIM language with the ROS middleware. The communication infrastructure of the integrated framework, graphically depicted in Figure 4.2, is based on ROS Bridge. This server is included in the ROS framework and provides a JSON API to ROS functionalities for external programs. This way, the ROS framework installed in a robot receives and executes commands on the physical components of the robot and gives feedback and sensor data.

The use of JSON enables the interoperability of ROS with most programming languages, including Java. As an example, we report in Figure 4.3 a message `pose` in the JSON format published on the ROS topic `/goal`, providing information for navigating a delivery robot to a given goal position. In our example, the goal is the position $(-0.21, 0.31)$.

X-KLAIM programs can indirectly interact with the ROS Bridge server, publishing and subscribing over ROS topics via objects provided by the Java library *java_rosbridge*.[1] In its own turn, *java_rosbridge* communicates with the ROS Bridge server via the WebSocket protocol through the Jetty web server.[2]

ROS permits checking the execution of the code generated from an X-KLAIM program by means of the Gazebo[3] simulator. Gazebo [74] is an open-source simulator of robot behaviors in complex environments that is based on a robust physics engine and provides a high-quality 3D visualization of simulations. Gazebo is fully integrated with ROS; in fact, ROS can interact with the simulator via the publish-subscribe communication mechanism of the framework. The use of the simulator is not mandatory when ROS is deployed in real robots. However, even in such a case, the MRS software design activity may benefit from using a simulator to save time and reduce development costs.

Since the X-KLAIM compiler generates plain Java code, which depends only on KLAVA and a few small libraries, deploying an X-KLAIM application can be made by using standard Java tools and mechanisms. It is enough to

[1] `https://github.com/h2r/java_rosbridge`
[2] Jetty 9: `https://www.eclipse.org/jetty/`
[3] `https://gazebosim.org/`

create a jar with the generated Java code and its dependencies (KLAVA and *java_rosbridge*), that is, a so-called "fat-jar" or "uber-jar". Such a jar file can be deployed to every physical robot where a Java virtual machine is already installed. Under that respect, X-KLAIM provides standard Maven artifacts and a plugin to generate Java code outside Eclipse, e.g., in a Continuous Integration server. Moreover, the dependencies of an X-KLAIM application, including *java_rosbridge*, are only a few megabytes, which makes X-KLAIM applications suitable also for embedded devices like robots.

4.2 The X-KLAIM approach at work on MRS scenarios

To illustrate the proposed approach, in this section, we show and briefly comment on a few interesting parts of implementing two warehouse scenarios[4] involving an MRS that manages the movement of items.

In Section 4.2.1, we present a simple warehouse scenario with an MRS composed of an arm robot and a delivery robot working together in an environment free of obstacles. Then, in Section 4.2.2, we show an enriched version of the scenario with multiple delivery robots and a more realistic warehouse environment, focusing on how the code of Section 4.2.1 can be reused and extended in only a few parts.

4.2.1 Simple warehouse scenario

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

In Figure 4.4, we show a part of the network for our scenario implementation. Each robot is rendered as an X-KLAIM node, whose name represents its locality (see Section 2.1.2).

Each node creates its process locally, representing the node/robot behavior, and executes it concurrently using the X-KLAIM operation `eval`. The delivery robot process behavior is parametric concerning the identifier and the sector. These parameters facilitate the reusability of the process, as shown in Section 4.2.2. The node `Environment` provides an interface with the simulated environment. In this scenario, it notifies the arm robot about the presence of items to be picked up and consumes them when delivered

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

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

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

   node Environment {...}
}
```

Figure 4.4: The X-KLAIM net of the simple warehouse scenario.

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

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

Figure 4.5: The process representing the arm robot behavior.

to the destination. While these actions are simulated here, in a real-world implementation of the scenario they might be performed by physical devices or human actors.

In this paper, the processes representing robot behavior have a common shape. As an example, we show the process of the arm behavior in Figure 4.5. The idea is that the behavior process defines the lifecycle while the actual implementation logic is delegated to another process (`PickAndReleaseOneItem`, in this case). The process responsible for implementing the logic is meant to be usable with different behaviors. In this example, all the behaviors are recursive. In fact, in Figure 4.5, after the execution of the implementation logic, the `ArmBehavior` evaluates another instance of the behavior. Hence, before starting a new instance of the behavior, it has to coordinate with `PickAndReleaseOneItem`. The latter is expected to put a tuple in the local tuple space with an agreed string. To avoid possible spelling mistakes when using constants in tuples, we define the constants in a Java class `ArmConstants`. Note that, thanks to the integration with Java (see Section 2.1.2), X-KLAIM can use Java constants with standard "import static" mechanisms. Recall that `eval` spawns another con-

current process and is a non-blocking operation. Thus, the currently running behavior process `ArmBehavior` terminates after starting another instance of itself.

In Figure 4.6, we show the code of the process `PickAndReleaseOneItem`. This process waits for a tuple with information concerning an item available for delivery (in our implementation this is provided by the node `Environment` of Figure 4.4). Then, it defines a few constants representing trajectories. Trajectories are implemented with plain Java objects and contain a few double numbers corresponding to physical points in the scenario, some of which depend on the item coordinates. We do not show them here because they are not relevant to the aim of this section. The actual logic is implemented by relying on a few reusable processes: `MoveArm` and `UseGripper`, whose names and usages should be self-explanatory. Both processes are parameterized with a trajectory structure, that represents the actual movement, and are started, once again, with `eval`. They notify the completion of their task by outputting a particular tuple which is consumed by the process `PickAndReleaseOneItem` before starting the execution of the next process. On its termination, the process outputs the tuple with `IS_IN_THE_INITIAL_POSITION`, expected by `ArmBehavior` (Figure 4.5), to notify that it has finished its tasks.

In Figure 4.7, we show the code of the process `MoveArm`. Like `UseGripper` (which we do not show here), `MoveArm` relies on the ROS Bridge. As already discussed in Section 4.1, the execution of an X-KLAIM robotics application requires the ROS Bridge server to run, providing a WebSocket connection at a given URI. The URI of the ROS Bridge WebSocket is one of the Java constants we defined. In the code of our example application, we consider the ROS Bridge server running on the local machine (0.0.0.0) at port 9090.

The process `MoveArm` connects to the ROS Bridge and initializes a publisher for the topic related to the control of arm movements. The process defines the trajectory for the arm movement and publishes it. Then, the process uses the Java API provided by *java_rosbridge* for subscribing to a specific topic (we refer to *java_rosbridge* documentation for the used API). The last argument of `bridge.subscribe` is a lambda expression (see Section 2.1.2). The lambda expression will be executed when an event for the subscribed topic is received. In particular, the lambda expression reads some data from the event (in JSON format) concerning the "positions". ROS dictates the JSON message format. To access the contents, we use the standard Java API (`data` is of type `JsonNode`, from the *jackson-databind* library). The lambda expression calculates the delta between the actual joint positions and the destination positions to measure the arm movement's completeness. The `if` determines when the arm has completed the rotation movement ac-

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

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

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

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

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

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

   out(ITEM_READY_FOR_DELIVERY,sector)@self

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

   in(DELIVERY_ROBOT_ARRIVED)@self

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

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

   out(GRIPPER_OPENED,itemId,itemType)@self

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

   out(IS_IN_THE_INITIAL_POSITION)@self
}
```

Figure 4.6: The process with the logic of the arm robot.

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

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

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

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

Figure 4.7: The process for moving the robotic arm.

cording to a specific tolerance. When that happens, the lambda expression notifies that the task is completed. This is achieved by inserting a particular tuple in the local tuple space. The process in Figure 4.6 will consume this tuple and will go on by spawning the process for the next movement. Finally, we can unsubscribe from the topic to stop receiving notifications from the ROS Bridge.

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

The implementation of the delivery robot (see the node `DeliveryRobot` in Figure 4.4) follows a similar strategy. The `DeliveryRobotBehavior` is similar to the behavior of Figure 4.5, and we do not show it here.

The process with the logic of the delivery robot is shown in Figure 4.8. Similarly to `PickAndReleaseOneItem` of Figure 4.6, this process delegates the physical actions to reusable processes that use the ROS Bridge: `WaitForItem` (which we do not show here) and `MoveTo`, which is parameterized over the target destination.

We show `MoveTo` in Figure 4.9, including the parts that deal with mathematical computations concerning the currently read position. The parts for using the ROS Bridge and coordinating through the tuple space are similar to the ones of `MoveArm` of Figure 4.7. The delivery robot navigation in this process is based on a proportional control technique that adjusts the robot's linear and angular velocities depending on its current position and orientation relative to the target. It calculates the heading error, which is the difference between the angle the robot is currently facing and the angle it needs to reach the target, as well as the distance error. The linear and angular velocities are then adjusted based on proportional gain, where the distance error is proportional to the linear velocity and the heading error is proportional to the angular velocity. This approach is simple but effective for navigating to a specific point in an environment free of obstacles. However, it may not be suitable for more complex scenarios that include static and dynamic objects. In those cases, a more advanced navigation system, like *2D navigation stack* provided by ROS, may be required, as shown in Section 4.2.2.

To recap, we propose an approach that clearly separates the responsibil-

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

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

   out(DELIVERY_ROBOT_ARRIVED)@Arm

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

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

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

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

   out(AVAILABLE_FOR_DELIVERY)@self
}
```

Figure 4.8: The process with the logic of the delivery robot.

ities among different processes, which can be seen as different logical layers:

- We have a process for the high-level behavior of the robot, like `ArmBehavior` of Figure 4.5, which only takes care of establishing the lifecycle of the robot;
- We have a process for implementing the main logic of the robot, like `PickAndReleaseOneItem` of Figure 4.6, which relies on the reusable building blocks of parameterized processes that are responsible for using the ROS Bridge for communicating with the robot's physical parts;
- These latter processes implement the main physical actions. Still, they are reusable thanks to their parameterization. For example, we have a single process representing the "movement of the arm"; according to the parameter, the process will go down, get up, rotate, etc. This is quite different from the approach presented in [21], where we used a different

```
proc MoveTo(String robotId, Double x, Double y) {
    val local = self
    // set the tolerance for distance and angle error
    val distanceTolerance = 0.1;
    val angleTolerance = 0.1;
    // connect to the ROS bridge
    val bridge = new XklaimToRosConnection(ROS_BRIDGE_SOCKET_URI)
    // initialize a publisher for the topic related to the control of the robot's wheels
    val pub = new Publisher("/" + robotId + "/cmd_vel", "geometry_msgs/Twist", bridge)
    // create the message for sending velocity commands
    val vel_msg = new Twist()
    val PI = 3.141592654;
    // gain K used to calculate the linear velocity
    val double K_l = 0.5;
    // gain K used to calculate the angular velocity
    val double K_a = 0.5;
    // subscribe to the robot's odometry sensor data
    bridge.subscribe(
        SubscriptionRequestMsg.generate("/" + robotId + "/odom").setType("nav_msgs/Odometry").setThrottleRate(1).
            setQueueLength(1),
        [ data, stringRep |
            var mapper = new ObjectMapper();
            var JsonNode rosMsgNode = data.get("msg");
            var Odometry odom = mapper.treeToValue(rosMsgNode, Odometry);
            var double currentX = odom.pose.pose.position.x;
            var double currentY = odom.pose.pose.position.y;
            var angle = new EulerAngles(odom.pose.pose.orientation)
            var currentTheta = angle.yaw
            // calculate the error in heading and distance b
            var deltaX = x - currentX;
            var deltaY = y - currentY;
            var angular = Math.atan2(deltaY, deltaX)
            var headingError = angular - currentTheta
            if (headingError > PI) {
                headingError = headingError - (2 * PI); // wrap angle to [-PI,PI]
            }
            if (headingError < -PI) {
                headingError = headingError + (2 * PI); // wrap angle to [-PI,PI]
            }
            var distance = Math.sqrt(Math.pow((x - currentX), 2) + Math.pow((y - currentY), 2))
            if ((distance > distanceTolerance)) {
                if (Math.abs(headingError) > angleTolerance) {
                    vel_msg.linear.x = 0.0;
                    // calculating angular velocity to correct heading
                    vel_msg.angular.z = K_a * headingError;
                } else {
                    vel_msg.linear.x = K_l * distance;
                    vel_msg.angular.z = 0.0;
                }
                pub.publish(vel_msg)
            } else {
                // the robot reached the goal and stops moving
                vel_msg.linear.x = 0
                vel_msg.angular.z = 0
                pub.publish(vel_msg)
                out(MOVE_TO_COMPLETED)@local
                bridge.unsubscribe("/" + robotId + "/odom")
            }
        ]
    )
}
```

Figure 4.9: The process for moving the delivery robot.

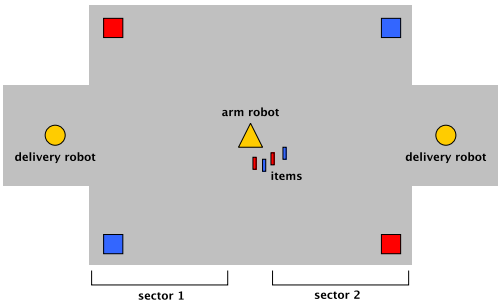


Figure 4.10: Enriched warehouse scenario.

X-KLAIM process for every single movement, e.g., `GetDown`, `GetUp`, `Rotate`, etc. This led to many processes with some code duplicated across those processes that might become hard to read and maintain.

We believe this clear separation of responsibilities enhances our code's re-usability, readability, and maintainability. This also allows us to follow an incremental approach, as shown in this section: we start focusing on a smaller problem/scenario (one single delivery robot, one single item type, no obstacles); then, we scale to a more complex one (two delivery robots, different item types, presence of obstacles), by reusing most of the code of the simple scenario and extending/modifying only a few processes for the goal of the advanced scenario, as we show in the next Section 4.2.2.

4.2.2 Enriching the warehouse scenario

In this section, we present an evolution of the simple scenario of Section 4.2.1. Most of the code is the same as in Section 4.2.1, and in this section, we focus on the parts that must be adapted for the evolved scenario.

As shown in Figure 4.10, in this scenario, the MRS is composed of an arm robot and two delivery robots, and the warehouse is divided into two sectors, each one served by a delivery robot. Similarly to the previous scenario, the arm robot, positioned in the center of the warehouse, picks up one item at a time from the ground, calls the delivery robot assigned to the item's sector, and releases the item on top of the delivery robot. The latter delivers the item to the appropriate delivery area, which depends on the item's color, and then becomes available for a new delivery. In addition, delivery robots must deal with obstacles (e.g., pallets) that are in the warehouse.

In Figure 4.11, we show a part of the network for this scenario implementation. The network is similar to the one of Figure 4.4 since we reuse the processes for the behavioral parts. It mainly differs for the fact that, since in this scenario the destination coordinates depend on the item type (i.e.,

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

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

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

   node Environment   {...}
}
```

Figure 4.11: The X-KLAIM net of the enriched warehouse scenario.

color), before activating the `DeliveryRobotBehavior` processes, the delivery robot nodes insert in the local tuple space the `ITEM_DESTINATION` tuples that define the mapping from item type to destination coordinates.

We have to adapt the process `DeliveryOneItem` as shown in Figure 4.12. Comparing the code of this process with the corresponding one of the first scenario (Figure 4.8), we see that it retrieves the destination coordinates at run-time from the local tuple space through the `ITEM_DESTINATION` tuples. Notably, the two robots deliver items of the same type to different destinations.

We also have to adapt the process `MoveTo` as shown in Figure 4.13. This process is much simpler in this scenario, even if it relies on a more sophisticated navigation system leveraging the navigation stack packages provided by ROS. Specifically, we exploit the topic `move_base_simple` to communicate with the ROS node of the navigation system, which accepts messages containing the goal coordinates. This node is an integral component of the navigation stack. It is responsible for linking the global planner with the local planner to determine the robot's trajectory while avoiding obstacles. The global planner generates the path as a sequence of waypoints the robot

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

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

   out(DELIVERY_ROBOT_ARRIVED)@Arm

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

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

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

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

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

   out(AVAILABLE_FOR_DELIVERY)@self
}
```

Figure 4.12: The process with the logic of the delivery robot in the enriched warehouse scenario.

must follow. The local planner generates the low-level plan to move from one waypoint to the next. This mechanism relies on a map of the environment and localization facilities to achieve this. To determine the completion of the robot's movement, the process `MoveTo` subscribes to the topic `amcl_pose`, which provides an estimate of the robot's pose in the given map using the "Adaptive Monte Carlo Localization" algorithm (AMCL).

The other processes for implementing this scenario are the same as the corresponding ones of the simple scenario. As anticipated in Section 4.2.1, having separated responsibilities in reusable processes allowed us to follow an incremental approach: we reused most of the code of the simple scenario, and we had to modify/replace only a few processes according to the requirements of this scenario. The screenshot in Figure 4.14 shows this scenario in execution. On the left, the Eclipse IDE with our X-KLAIM code is shown (see the logged messages on the Console). On the right, the Gazebo simu-

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

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

Figure 4.13: The process for moving the delivery robot in the enriched warehouse scenario.

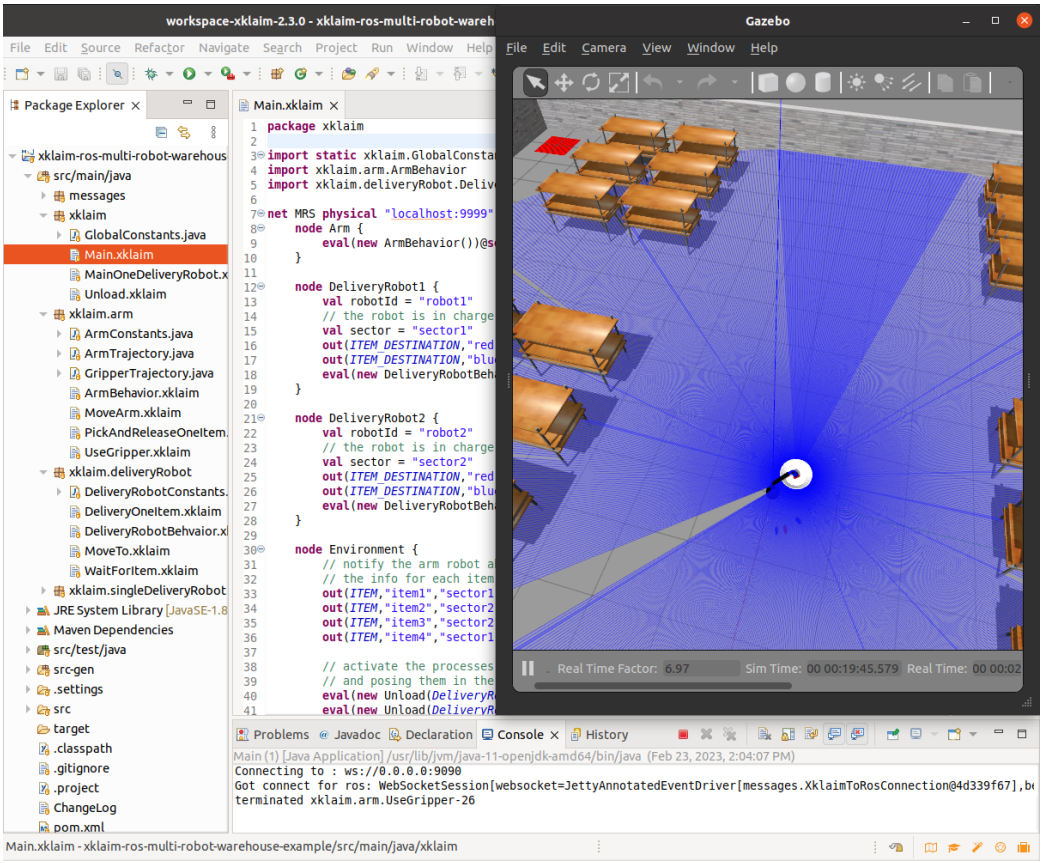


Figure 4.14: Execution of the X-KLAIM robotics application of the enriched warehouse scenario.

lator is shown, visualizing the arm in the center, ready to drop the item on top of the delivery robot's white plate.

4.2.3 Other Scenarios

Rescue Robot

The rescue robot scenario[5] 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.

Robot Collective Search

In the robot collective search scenario[6], a team of four robots works together to locate each uniquely colored flag within a designated area, despite not having prior knowledge of their positions. The flags are placed randomly throughout the environment. As they navigate through the environment, the robots must avoid the obstacles and continuously gather information about their proximity to the flags. When a robot detects a flag nearby, based on a predefined distance threshold, it notifies the other robots about the discovery. Once all the flags have been found, the robots cease their movements and the task is considered complete.

Formation Pattern

In the formation pattern scenario[7], we consider a team of autonomous, anonymous, and identical mobile robots forming various patterns, such as line, circle, and grid formations. For the line formation, the robots arrange themselves in a straight line, maintaining a specific distance from each other.This formation can be useful in scenarios where the robots need to traverse narrow passages or when they are required to maintain a clear line of sight. In the circle formation, the robots position themselves equidistantly around a

[5]The full source code of the rescue robot scenario can be found at `https://github.com/LorenzoBettini/xklaim-ros-example`.

[6]The full source code of the robot collective search can be found at: `https://github.com/khalidbourr/Collective_Search_MRS`.

[7]The full source code of the formation pattern can be found at `https://github.com/khalidbourr/xklaim-ros-swarm-robots`.

central point, formation a circle. This formation can be beneficial in scenarios where the robots need to surround an area of interest or maintain a perimeter. The grid formation involves the robots being organized in a grid pattern, where each robot occupies a cell within the grid. This formation can be advantageous in scenarios where the robots are required to cover a large area efficiently, such as during search and rescue missions or environmental monitoring tasks.

In each formation type, the robots communicate with their neighbors, share their positions, and adjust their positions based on the desired formation pattern. These formations enable the team of robots to adapt to different tasks and environmental constraints effectively.

4.3 Experimental Evaluation

In this section, we illustrate the experiments we carried out to determine the impact of our approach on MRS performance. The experiments are designed to provide a comparison of time and memory performance of our implementation of the MRS of the warehouse scenarios based on Java code and ROS Bridge against the traditional ROS implementation based on Python code.[8] To this aim, we exploit the warehouse scenario in Section 4.2.2, evaluating the overall execution time in milliseconds and memory consumption for each robot activity while using our solution and the traditional one. To guarantee consistency, the same hardware/software[9], input, and tasks are used in both environments. To account for variance, we ran the same experiments (i.e., the Java-based and the Python-based code) 30 times and averaged the results[10].

4.3.1 Time consumption

We determined the average completion time for each robot activity in milliseconds using our implementation, based on *java_ rosbridge*, and the Python one, based on *rospy*.[11] We discuss here the results of the experiments con-

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

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

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

[11]`http://wiki.ros.org/rospy`

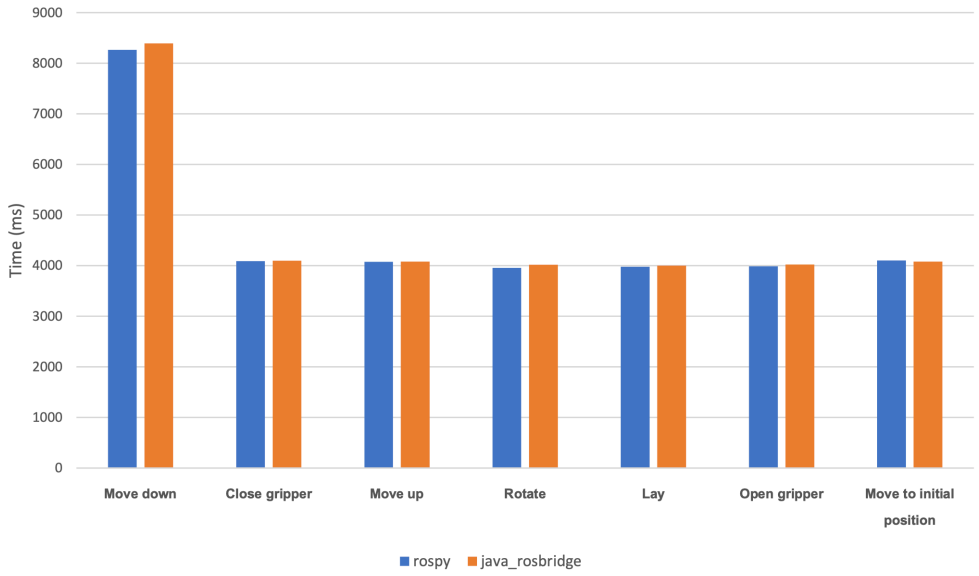


Figure 4.15: Time consumption

cerning the arm robot's activities; the ones concerning the delivery robots returned similar results. Each activity uses a publisher for sending the message that starts the enactment of the activity and a subscriber for receiving sensor data.

Figure 4.15 shows the time consumption for the activities of the arm robot. Each activity corresponds to an X-KLAIM process in our approach (except for "Move down" that corresponds to two executions of `MoveArm`, with arguments `HALF_DOWN` and `COMPLETE_DOWN`, respectively) and a class in the Python implementation. For example, the "Rotate" activity takes an average of 3955 milliseconds in Python and 4019 milliseconds in Java to execute. The time consumption is similar for activities using different topics; e.g., "Open gripper" takes 3986 milliseconds in Python and 4021 milliseconds in Java. The experiment results indicate that the Java program has a slightly greater latency than the Python version. This is a consequence of the serialization and deserialization of messages, network overhead, and connection with the ROS Bridge server via the WebSocket protocol. However, in the case under evaluation, the average delay difference between the two setups is at most 200 milliseconds. Therefore, the overhead introduced by our solution does not significantly affect the mission of the considered MRS.

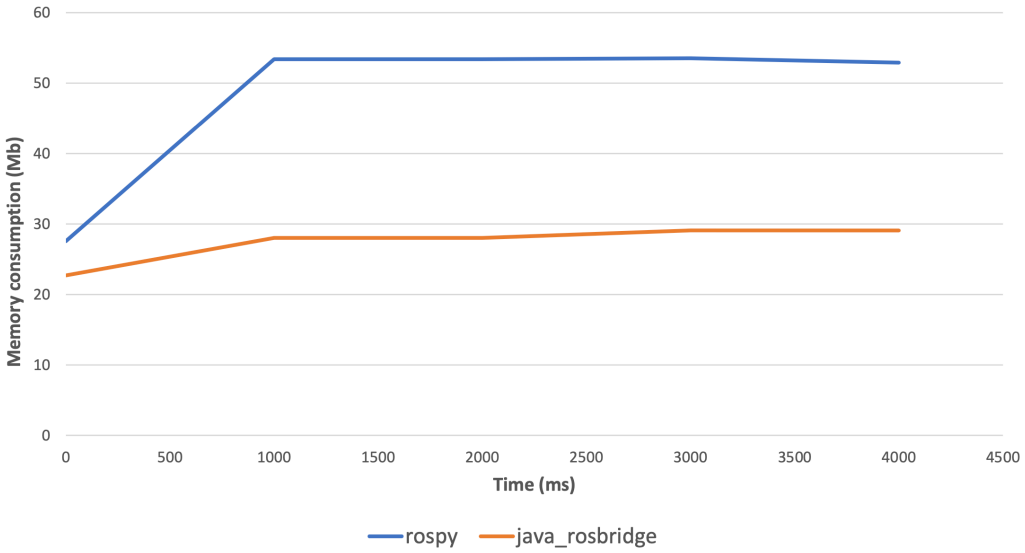


Figure 4.16: Memory consumption of the Rotate activity.

4.3.2 Memory consumption

To measure the memory consumption, we employed two different tools for performance profiling and monitoring: VisualVm[12] for Java code, and Memray[13] for Python.

We show here the results of the experiments on one robot's activity, namely "Rotate". As shown in Figure 4.16, after the start-up phase of the activity, the heap used by Python is more than 50MB, while Java uses almost 30MB. The trend for the other activities is quite similar. Even if identifying the cause of this difference is not relevant to our investigation, we think that, in this experiment, the difference might be attributed to more efficient automatic memory management in Java compared to Python. It is worth noting that our approach also requires the execution of the ROS Bridge server, which uses around 135KB of memory and, hence, does not significantly affect the overall memory cost.

4.4 Discussion and Related work

Over the last few years, researchers have attempted to define notations closer to the robotics domain to raise the abstraction level for enabling automated code generation, behavior analysis, and property verification (e.g., safety and performance). This section reviews several high-level languages and

[12] `https://visualvm.github.io`
[13] `https://github.com/bloomberg/memray`

frameworks for modeling, designing, and verifying ROS-based applications and some languages for coordinating collaborative MRSs. We summarize in Table 4.1 our considerations and comparison with the languages more strictly related to ours.

DSL	*Formal language*	*High-level language*	*Multi-robots*	*Heterogenous robots*	*Coordination*	*Decentralized coordination*	*Open-endedness*	*Compiler*	*IDE*	*ROS*
ART2ool [75]		✓						✓		✓
ATLAS [76]		✓	✓		✓			✓		✓
BRIDE [77]		✓						✓	✓	✓
CommonLang [78]		✓						✓		✓
Drona [79]	✓	✓	✓		✓	✓		✓		✓
FLYAQ [80]		✓	✓							✓
Hyperflex [81]		✓						✓	✓	✓
ISPL [72]	✓	✓	✓		✓	✓		✓		
Koord [82]	✓	✓	✓	✓	✓	✓		✓		✓
PROMISE [83]		✓	✓							✓
RobotChart [84]	✓	✓						✓	✓	
ROSBuzz [85]	✓	✓	✓	✓	✓	✓		✓	✓	✓
RSSM [86]	✓							✓		✓
SCEL [71]	✓	✓	✓	✓	✓	✓	✓	✓		
X-Klaim	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.1: Features comparison of the related works.

High-level languages and frameworks Many DSLs for component-based modeling of robotic systems are based on UML and target mostly the architectural aspect of robotic applications, e.g., RobotML [1], V^{3}CMM [87], BRICS [88], RoboChart [84], and SafeRobots [89]. Some of them can be used to build ROS-based systems by either supporting a direct translation, e.g., Hyperflex [81], or serving as a base for other platforms. For example, in BRIDE [77], which relies on BRICS, the components are modeled using UML and converted to ROS meta-models to generate runnable ROS C++ code. Additional meta-models (i.e., deployment meta-model and experiment meta-model) for rapid prototyping component-based systems are provided in [90]. UML has also been used to model and design robotic tasks and missions, e.g., Art2ool [75] supports the development cycle of robotic arm tasks in which atomic tasks are abstracted with UML class diagrams. Textual languages, e.g., CommonLang [78], are another type of language used to model robotic systems. For example, in [91], a DSL based on the Python language can be used interactively, through the Python command-line interface, to create brand new ROS nodes and reshape existing ROS nodes by wrapping their communication interfaces.

Some other contributions, to some extent, allow for the verification of ROS-based systems. ROSGen [92] takes a specification of a ROS system architecture as an input and generates ROS nodes as an output. Using the theorem prover Coq, the generation process is amenable to formal verification. DeROS [93] permits describing robot's safety rules (and their related corrective actions) and automatically generating a ROS safety monitoring node by integrating these rules with a run-time monitor. Another framework for run-time verification of ROS-based systems is described in [94], which allows generating C++ code for a monitoring node from user-defined properties specified in terms of event sequences. In [95], robot systems are modeled as a network of timed automata that, after verification in Uppaal,[14] are automatically translated into executable C++ code satisfying the same temporal logic properties as the model. Finally, RSSM [86] enables modeling of multi-agent robot's activities using Hierarchical Petri Nets. After checking for deadlock absence on the model, RSSM can generate C++ code for ROS packages automatically.

The approaches mentioned above have not been applied to such complex systems as MRSs, and some are not even suitable for such systems. Very few high-level languages for MRSs have been proposed. For example, FLYAQ [80] is a set of DSLs based on UML to specify the civilian missions for unmanned aerial vehicles. This work is extended in [96] for enabling the use of a declarative specification style, but it only supports homogeneous robots. ATLAS [76], which also provides a simulator-based analysis, takes a

[14] `https://uppaal.org/`

step further towards coordinating MRSs but only supports centralized coordination. PROMISE [83] allows specifying the missions of MRSs using Linear Temporal Logic operators for composing robotic mission patterns. Finally, RMoM [97] first allows using a high-level language for specifying various constraints and properties of ROS-based robot swarms with temporal and timed requirements and then automatically generating distributed monitors for their run-time verification.

Languages for coordination Coordination for MRSs has been investigated from several diverse perspectives. Nowadays, many techniques can be used to orchestrate the actions and movements of robots operating in the same environment [10, 4]. Designing fully-automated and robust MRSs requires strong coordination of the involved robots for autonomous decision-making and mission continuity in the presence of communication failures [61]. Several studies recommend using indirect communication to cut implementation and design costs usually caused by direct communication. Indirect communication occurs through a shared communication structure that each robot can access in a distributed concurrent fashion. Some languages providing communication and coordination primitives suitable for designing robust MRSs are reviewed in [5]. In ISPL [72], communication is obtained as an indirect result of synchronizing multiple labeled transition systems on a specific action. In SCEL [71], a formal language for the description and verification of collective adaptive systems, communication is related to the concept of knowledge repositories, represented by tuple spaces. In Buzz [98], a language for programming heterogeneous robot swarms, communication is implemented as a distributed key-value store. For this latter language, integration with the standard environment of ROS has also been developed, which is named Rosbuzz [85]. Unlike X-KLAIM, however, Rosbuzz does not provide high-level coordination primitives, robots' distribution is not explicit, and permits less heterogeneity. Drona [79] is a framework for distributed drones where communication is somehow similar to the one used in ISPL. Koord [82] is a language for programming and verifying distributed robotic applications where communication occurs through a distributed shared memory. Unlike X-KLAIM, however, robot distribution is not explicit, and open-endedness is not supported. Finally, in [99], a programming model and a typing discipline for complex multi-robot coordination are presented. The programming model uses choreographies to compositionally specify and statically verify message-based communications and jointly executed motion between robotics components in the physical space. Well-typed programs, which are terms of a process calculus, are then compiled into programs in the ROS framework.

Chapter 5

X-Klaim Mission Specification Patterns for ROS-Based Robots Systems

Efficient and effective mission planning and execution play a crucial role in the successful operation of robots in various environments. By leveraging the capabilities of the X-Klaim programming language, we can design and implement complex robotic tasks that are both scalable and maintainable. In this chapter, we will explore a comprehensive set of core movement patterns, following the approach outlined in [100], which offers a catalog of 22 mission specification patterns for mobile robots. In our work, we focus on a set of core movement patterns and their corresponding X-Klaim implementation. Serving as fundamental building blocks for creating and executing diverse robotic missions, these patterns provide a structured approach to mission planning, laying the groundwork for the development for more robust, scalable, and reusable robotic solutions developed using X-Klaim.

The complete X-Klaim implementations for all patterns can be found in the GitHub[1] repository.

5.1 Core Movement Patterns for ROS-based Robots

Core movement patterns form the foundation for designing robotic missions, providing a structured approach to mission planning and execution. By combining and adapting these patterns, we can create more complex tasks and behaviors suitable for a wide range of robotic applications. The core

[1] https://github.com/khalidbourr/xklaim-patterns-robot-mission.

movement patterns can be broadly divided into two categories: Coverage (`Visit`, `Sequenced Visit`, `Ordered Visit`, etc.) and Surveillance (`Patrolling`, `Ordered Patrolling`, etc.). Coverage patterns focus on ensuring that the robot visits all specified locations, while Surveillance patterns emphasize the continuous monitoring of specific areas. These building blocks are designed to extend and enhance the capabilities of the `MoveTo` action, described in Section 4.2.2. By building upon `MoveTo` action, these patterns integrate and coordinate robot movement more effectively, allowing for a seamless transition between tasks and increased mission flexibility. As a result, these extended building blocks enable more advanced mission specifications, making it easier to tailor robotic missions to the specific needs of various applications. In this section, we provide a description and an example for each pattern, illustrated with a graph. We focus on detailing the X-KLAIM implementations of the `Visit` and `Patrolling` patterns, which serve as representative examples for the coverage and surveillance categories, respectively. The logic and structure of other patterns are quite similar to the ones presented here, making it easier for readers to understand their implementation based on the provided examples.

5.1.1 Visit

Description: The robot must visit a set of locations in an unspecified order.
Example: A robot must visit locations l_1,l_2, and l_3 but can do so in any order, a possible trace is depicted in Figure 5.1, where l_x[2] is a location different from l_1,l_2, and l_3.

Figure 5.1: An Example of a Visit Pattern Trace

The implementation of the pattern, illustrated in Figure 5.2, begins by obtaining the number locations from the input list and generating a random trace using the `createRandom` function from the `TraceGenerator` class. This trace determines the order in which the robot visits the locations. A

[2]Notably, l_x is an intermediary location that the robot may traverse, distinct from the predefined set of target locations. l_x is dynamically determined based on the robot's underlying movement algorithms and the state of its current environment, including variables such as obstacles or other entities in the robot's path. As such, l_x is not a static location and may vary during runtime, reflecting changes in the robot's trajectory or environmental conditions.

list, named `visitedLocations`, is created to track the locations visited by the robot. The process iterates through the trace, retrieves each location, and adds its name to the `visitedLocations` list while printing a message indicating the robot's movement. To move the robot to the desired location, the `MoveTo` process is called with the robot's ID and the target location's coordinates. The process then waits for `MOVE_TO_COMPLETED` tuple to ensure the robot has arrived at the location. Once all the locations are visited, the process outputs the tuple with `VISIT_COMPLETED` and prints the list of visited locations.

```
/**
 * Visit a set of locations in an unspecified order.
 *
 * @param robotId   the ID of the robot
 * @param locations the list of locations to visit
 */
proc Visit(String robotId, List<Location> locations) {
    // Get the number of locations
    val n = locations.size()

    // Generate a random trace
    val trace = TraceGenerator.createRandom(n)

    // Create a list to store the visited locations
    val visitedLocations = new ArrayList<String>()

    // Visit the locations according to the trace
    for (i : trace) {
        // Get the location to visit
        val location = locations.get(i)

        // Print a message indicating the location the robot is moving to
        println("Moving to location: " + location.name)

        // Move to the location and wait for completion
        eval(new MoveTo(robotId, location.x, location.y))@self
        in(MOVE_TO_COMPLETED)@self
        // Add the location to the visited locations list
        visitedLocations.add(location.name)
    }

    out(VISIT_COMPLETED)@self
    // Print the visited locations list
    println("Visited locations: " + visitedLocations)
}
```

Figure 5.2: X-KLAIM implementation of the Visit pattern

The `createRandom` method, shown in Figure 5.3, is responsible for generating a random trace for visiting the locations. It uses a set called `uniqueNumbers` to ensure that each location is visited at least once. The generated trace is stored in the `trace` list. With the help of a random number generator, we add unique numbers to the trace list until it reaches the desired size. Finally, we convert the trace list into an integer array and return it.

```
    public static int[] createRandom(int n) {
        // Set to keep track of unique numbers
        HashSet<Integer> uniqueNumbers = new HashSet<>();
        // List to store the resulting trace
        List<Integer> trace = new ArrayList<>();
        // Random number generator
        Random rand = new Random();
        // Initialize the last added number to an invalid value
        int lastAdded = -1;

        while (uniqueNumbers.size() < n || trace.size() < 2 * n) {
            int nextValue = rand.nextInt(n);
            // If the generated number is not equal to the last added number, add it to the trace
            if (lastAdded != nextValue) {
                trace.add(nextValue);
                lastAdded = nextValue;
                uniqueNumbers.add(nextValue);
            }
        }

        // Convert the trace list to an int array and return it
        return trace.stream().mapToInt(i -> i).toArray();
    }
}
```

Figure 5.3: X-KLAIM implementation of Visit pattern

5.1.2 Sequenced Visit

Description: The robot must visit a set of locations in a specified order.
Example: A robot must visit locations l_1,l_2, and l_3 in that order, a possible trace is depicted in Figure 5.4.

Figure 5.4: An Example of a Sequenced Visit Pattern Trace

The implementation specifics are not detailed here due to their similarity to the `Visit` pattern previously discussed, with the main distinction being that this pattern relies on the `createSequenced` method to generate a visit trace in a specific order.

5.1.3 Ordered Visit

Description: While similar to the `Sequenced Visit` pattern in its requirement for the robot to visit a set of locations in a specific order, this pattern imposes additional restrictions. Unlike the `Sequenced Visit` patterns, which permits the robot to visit a location later in the sequence before

its predecessor, this pattern strictly forbids this. the robot can visit other locations, but it must strictly adhere to the order for the specified locations. **Example:** A robot must visit locations l_1,l_2, and l_3 in that order but can visit other locations as necessary, a possible trace is depicted in Figure 5.5.

Figure 5.5: An Example of an Ordered Visit Pattern Trace

The `Ordered Visit` pattern's implementation revolves around the `createOrdered` method. This method builds upon the logic of `createSequenced` but incorporates an additional to enforce strict adherence to the sequence order. It ensure no location appearing later in the sequenced is visited before its predecessor.

5.1.4 Strict Ordered Visit

Description: The robot must visit a set of locations in a specific order, and cannot visit any location more than once before visiting the next location in the sequence.
Example: A robot must visit locations l_1,l_2, and l_3 in that order, without revisiting any of the locations before completing the sequence, a possible trace is depicted in Figure 5.6.

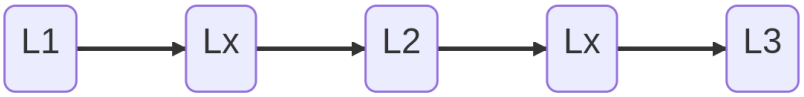


Figure 5.6: An Example of a Strict Ordered Visit Pattern Trace

For the `Strict Ordered Visit` pattern, the implementation employs the `createStrictOrdered` method. This method not only enforces the strict sequence of visits but also precludes any predecessor location from being visited multiple times before its successor. This added rule allows for a more granular control of the robot's visiting sequence.

5.1.5 Fair Visit

Description: The robot must visit a set of locations, ensuring that the difference in the number of visits to each location is at most one.
Example: A robot must visit locations l_1,l_2, and l_3, making sure that each location is visited an equal number of times or with at most one difference, a possible trace is depicted in Figure 5.7.

Figure 5.7: An Example of a Visit Pattern Trace

The implementation of `Fair Visit` pattern is facilitated by the `createFair` method. Initially, it generates a random trace, then verifies that each location is visited an equivalent number of times, ensuring the visitation discrepancy between any two locations is at most one.

5.1.6 Patrolling

Description: The robot must continuously visit a set of locations but not in a particular order.
Example: A robot must continuously visit locations l_1,l_2, and l_3 in any order, a possible trace is depicted in Figure 5.8.

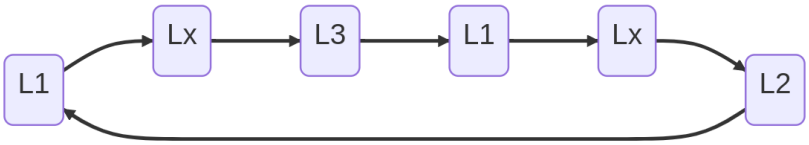


Figure 5.8: An Example of a Patrolling Pattern Trace

The `Patrolling` process, shown in Figure 5.9, designed for continuous surveillance of a set of locations without a specific order, accepts a robot ID, a list of locations, and the number of iterations as input parameters. It employs a loop that iterates a defined number of times to visit the given locations, ensuring the robot's ongoing patrol of the area. Within the loop, the `Visit` process handles the task of visiting each location in a randomly generated sequence. Upon completion, a `VISIT_COMPLETED` tuple is consumed,

confirming the conclusion of the visiting process before initiating the next iteration. This implementation allows the robot to consistently patrol the designated locations, fulfilling the mission requirement for the `Patrolling` pattern [3]. It is worth mentioning that, during each iteration of the loop, a fresh trace adhering to the visit pattern's requirements is generated, ensuring that the robot continues to patrol the area in varying sequences.

```
/**
   Keep visiting a set of locations, but not
   in a particular order.
 *
 *      @param robotId   the ID of the robot
 *      @param locations the list of locations to visit
 *      @param iterations the number of time the process should be executed
 */
proc Patrolling(String robotId,List<Location> locations, Integer iterations) {
   var count=0
   while(count<iterations){

      eval(new Visit(robotId, locations))@self
      in(VISIT_COMPLETED)@self
      count = count +1
   }
   out(PATROLLING_COMPLETED)@self
}
```

Figure 5.9: X-KLAIM implementation of Patrol pattern

5.1.7 Sequenced Patrolling

Description: The robot must continuously visit a set of locations in a specific order, but may visit other locations between the required ones.
Example: A robot must continuously visit locations l_1,l_2, and l_3 in that order but can visit other locations as necessary, a possible trace is depicted in Figure 5.10.

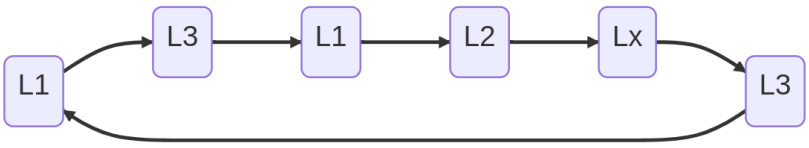


Figure 5.10: An Example of a Sequenced Patrolling Pattern Trace

[3]The randomized sequence of locations for each patrol cycle is generated dynamically within the loop, not pre-computed before the loop begins. This ensures a varying and unpredictable patrol pattern in each iterations.

In its implementation, this pattern achieves the desired result by continuously calling the `Sequenced Visit` process.

5.1.8 Ordered Patrolling

Description: The robot must continuously visit a set of locations in a specific order, but may visit other locations between the required ones.
Example: A robot must continuously visit locations l_1,l_2, and l_3 in that order, but can visit other locations as necessary, a possible trace is depicted in Figure 5.11.

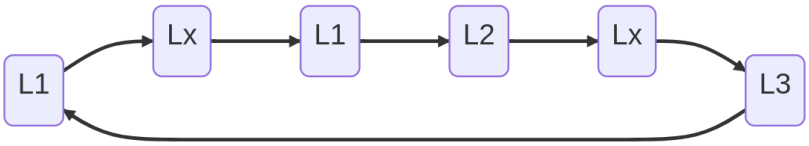


Figure 5.11: An Example of a Ordered Patrolling Pattern Trace

The pattern is implemented by invoking the `Ordered Visit` process in a loop.

5.1.9 Strict Ordered Patrolling

Description: The robot must continuously visit a set of locations in a specific order, and cannot visit any location more than once before visiting the next location in the sequence.
Example: A robot must continuously visit locations l_1,l_2, and l_3 in that order, without revisiting any of the locations before completing the sequence, a possible trace is depicted in Figure 5.12.

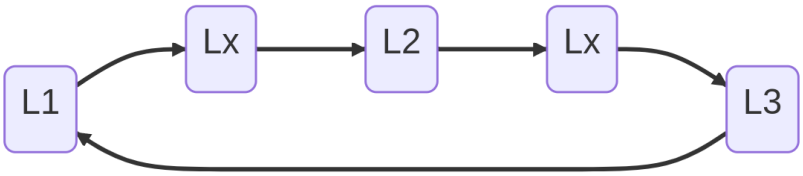


Figure 5.12: An Example of a Patrolling Pattern Trace

This pattern is executed by looping the `Strict Ordered Visit` process.

5.1.10 Fair Patrolling

Description: The robot must continuously visit a set of locations, ensuring that the difference in the number of visits to each location is at most one.
Example: A robot must continuously visit locations l_1,l_2, and l_3, making sure that each location is visited an equal number of times or with at most one difference. A possible trace is depicted in Figure 5.13.

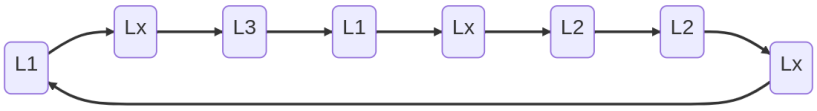


Figure 5.13: An Example of a Fair Patrolling Pattern Trace

In terms of implementation, this pattern involves continuously repeating `Fair Visit` process in a loop to ensure fairness in the frequency of visits.

5.2 Mission Scenarios Using Core Movement Patterns

In this section, we discuss the usability and adaptability of the core movement patterns presented earlier. The flexibility and composability of X-KLAIM allow for the combination of patterns to create complex missions. The language enables sequential actions by inserting a tuple indicating the completion of the action and consuming the tuple to move to next action as explained in Section 4.2.1, and running processes in parallel, supporting the creation of mission scenarios tailored to user's needs. We present three mission scenarios, highlighting the utilization of the core movement patterns and demonstrating the adaptability of X-KLAIM in creating missions. These examples serve to showcase how the patterns can be combined, extended, or modified to fit specific mission requirements.

5.2.1 Perimeter Surveillance Mission

In a perimeter surveillance mission, a robot is tasked with continually monitoring the perimeter of a specified area. It follows a predefined path, detecting and addressing any intrusions that might occur. This mission scenario showcases how the integration of core movement patterns and custom processes can result in a practical and efficient perimeter surveillance system.

The `PerimeterSurveillanceMission` process accepts the robot's ID, a list of perimeter locations, and a specific number of iterations as input parameters. Initially, the robots patrols the designated perimeter locations using the `Patrolling` process. Simultaneously, it runs a custom process, `DetectIntrusion`, designed to monitor to potential intrusions.

```
proc DetectIntrusion(String robotId){
   while(true){
      // Implement intrusion detection logic here
      // If an intrusion is detected, output the intrusion location

      if(intrusionDetected==true){
         out(INTRUSION_DETECTED, intrusionLocation)@self}
   }
}

proc HandleIntrusion(String robotId, Location intrusionLocation){
   // Implement intrusion handling logic here
   // For example; alert security, capture images, ect.

   out(INTRUSION_HANDLED)@self
}

proc PerimeterSurveillanceMission(String robotId,List<Location> perimeterLocations, Integer iterations) {
      eval(new Patrolling(robotId, perimeterLocations, iterations))@self
      eval(new DetectIntrusions(robotId))@self
      in(INTRUSION_DETECTED, Location intrusionLocation)@self

      eval(new Visit(robotId, intrusionLocation))@self
      in(VISIT_COMPLETED)@self

      eval(new HandleIntrusion(robotId, intrusionLocation))@self
      in(INTRUSION_HANDLED)@self

      eval(new PerimeterSurveillanceMission(robotId, perimeterLocations, iterations))@self
 }
```

Figure 5.14: X-KLAIM implementation of Perimeter Surveillance Mission

The `DetectIntrusion` process is responsible for implementing the intrusion detection logic. Upon detecting an intrusion, it outputs the location where the intrusion was detected. Following the detection of an intrusion, the `Visit` process is invoked, leading the robot to the intrusion location. After reaching the site of the intrusion, a custom process, `HandleIntrusion`, is executed to appropriately address the detected intrusion. Once the intrusion has been handled, the robot continues its surveillance duties by recursively calling the `PerimeterSurveillanceMission` process.

5.2.2 Coordinated Sector Coverage Mission

This multi-robot scenario exemplifies the efficacy and adaptability of core movement patterns in the context of a coordinated sector coverage mission.

In this mission, two robots are assigned to patrol different sectors of an area, with each sector containing four distinct locations, see Figure 5.15. By leveraging the `Patrolling` pattern, the robots can effectively cover their designated sectors, as depicted in Figure 5.16.

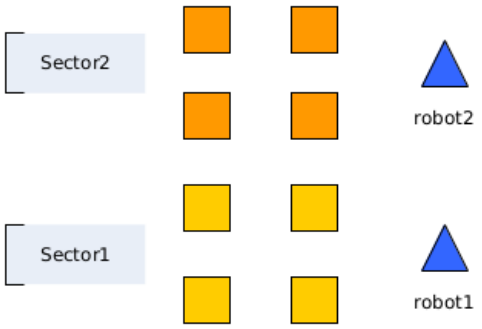


Figure 5.15: Sector Coverage Mission

```
net MRS physical "localhost:9999" {
    node Robot1 {
        val robotId = "robot1"
        val sector = #[l1,l2,l3,l4]
        val iterations = 10
        eval(new CoverageMission(robotId, sector, iterations))@self
    }
    node Robot2 {
        val robotId = "robot2"
        val sector = #[l5,l6,l7,l8]
        val iterations = 10
        eval(new CoverageMission(robotId, sector, iterations))@self
    }

}

/* Sector coverage process */
    proc CoverageMission(String robotId, List<Location> sector, Integer iterations) {
        eval(Patrolling(robotId, sector, iterations))@self
        in(PATROLLING_COMPLETED)@self
     }
```

Figure 5.16: X-KLAIM implementation of Sector Coverage Mission

In this implementation, the `Patrolling` process is employed, with each robot covering its assigned sector. The `CoverageMission` process accepts the robotId, a list of locations in the sector, and the number of iterations for the patrolling action as input parameters. By tailoring the mission to incorporate multiple robots and sectors, we can achieve efficient area coverage while demonstrating the versatility and flexibility of the core movement patterns in X-KLAIM.

5.2.3 Search and Rescue Mission

In a search and rescue mission, a robot (typically a drone) may be required to patrol a set of locations to search for survivors. Once a survivor is detected, another robot (a rescuer robot) is tasked with rescuing the survivor and transporting them to an extracting point, as depicted in Figure 5.17. The code in Figure 5.18 demonstrates the effective combination of core movement patterns and custom processes to achieve this complex objective. The `SearchMission` process takes the drone's ID, a list of patrol locations, and a frequency of patrolling as input parameters. The drone patrols the area and simultaneously detects survivors using the `Patrolling` and `DetectSurvivor` processes running in parallel. The detected survivor locations are outputted to the rescuer robot. On the other side, the `RescueMission` process handles the task of the rescuer robot. It takes the robot's ID and an extraction point location as input parameters. Upon receiving a survivor location from the drone, the rescuer robot proceeds to visit the survivor location using `Visit` process. It then executes the custom `RescueSurvivor` process for the located survivor, implementing the scenario-specific rescue logic. After rescuing a survivor, the robot transports them to the extraction point using the `Visit` process.

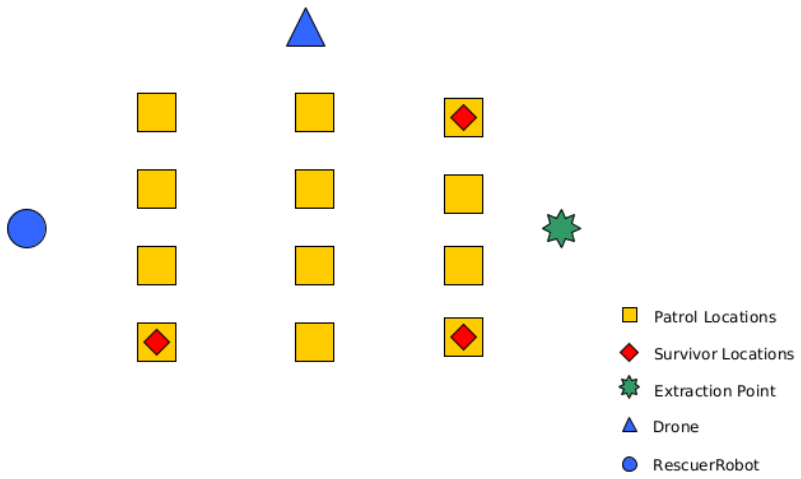


Figure 5.17: Search And Rescue Mission

By seamlessly integrating the core movement patterns (`Patrolling` and `Visit`) with custom processes (`RescueSurvivor` and `DetectSurvivor`), and by making effective use of parallelism and communication of these elements between nodes, the Search and Rescue Mission code exemplifies how a combination of these elements can achieve complex objectives such as patrolling, locating survivors, rescuing them, and transporting them to an extraction point.

```
net MRS physical "localhost:9999" {
    node Drone {
        val robotId = "DroneId" // Set drone id
        val patrolLocations = #[...] //  Set patrol locations
        val iterations = 10 // Set frequency of patrolling
        eval(new SearchMission(robotId, patrolLocations, iterations))@self
    }

    node RescuerRobot {
        val robotId = "RescuerId" // Set  rescuer id
        val extractionPoint = ... // Set extraction point location
        eval(new RescueMission(robotId, extractionPoint))@self
    }

    proc SearchMission(String robotId, List<Location> patrolLocations, Integer iterations) {
        eval(new Patrolling(robotId, patrolLocations, iterations))
        eval(new DetectSurvivor(robotId))@self
        in(PATROLLING_COMPLETED)@self
    }

    proc DetectSurvivor(String robotId) {
        // Implement survivor detection logic here

        out(SURVIVOR_LOCATION, survivorLocation)@RescuerRobot
    }

    proc RescueMission(String robotId, Location extractionPoint) {
        in(SURVIVOR_LOCATION, Location survivorLocation)@self

        eval(new Visit(robotId, survivorLocation))@self
        in(VISIT_COMPLETED)@self

        eval(new RescueSurvivor(robotId, survivorLocation))@self
        in(SURVIVOR_RESCUED)@self

        eval(new Visit(robotId, extractionPoint))@self
        in(VISIT_COMPLETED)@self

        eval(new RescueMission(robotId, extractionPoint))@self
    }

    proc RescueSurvivor(String robotId, Location survivorLocation) {
        // Implement rescue logic here

        out(SURVIVOR_RESCUED)@self
    }
}
```

Figure 5.18: X-KLAIM implementation of Search And Rescue Mission

5.3 Discussion

In this section, we elaborate on the ways our work builds upon and distinguishes itself from the prior work that presented a catalogue of 22 mission specification patterns for mobile robots [100]. While the earlier work primarily centers on providing a set of patterns expressed in LTL and CTL temporal logics, our work harnesses the power of X-KLAIM programming language to

design, implement, and coordinate complex robotic missions.

Process Coordination Our approach places significant emphasis on the coordination and communication between the various processes involved in robotic missions. Employing the X-KLAIM language enables us to facilitate seamless interaction between these processes, resulting in enhanced mission efficiency and effectiveness.

Multi-node Deployment By allowing mission patterns to be deployed across several robot nodes, our work expands the original catalogue's scope. This facilitates a more scalable and distributed approach to mission planning, paving the way for increasingly complex and coordinated missions.

Concurrent Execution We take advantage of the X-KLAIM language's inherent support for concurrent execution of processes. This capability enables our mission patterns to run simultaneously, improving overall mission performance and facilitating the excursion intricate, multi-robot missions.

Pattern Composability Our work augments the flexibility and adaptability of the core movement patterns by enabling their compositions and combination in diverse ways. This leads to the creation of complex missions tailored to specific user needs, making our approach highly versatile and customizable.

Extensibiilty and Adaptabilty The X-KLAIM language empowers users to extend and modify core movement patterns to suit particular mission requirements. This feature enables our work to address a wider range of mission scenarios and use case, yielding more robust, scalable, and reusable robotic solutions.

It is important to note that the core movement patterns have been rigorously tested in the Gazebo simulator. Nevertheless, the mission scenarios outlined in this chapter primarily serve as illustrations to showcase the pattern's versatility in various robotic tasks. These examples highlight the potential applications of the patterns within X-KLAIM, rather than representing actual deployed missions.

PART III

FROM MRSS MODELS TO CODE

Chapter 6

Multi-Robot Mission Modeling using BPMN

In this chapter, we introduce a novel approach for high-level modeling of cooperative behavior of MRSs by utilizing BPMN 2.0 collaboration diagrams. The proposed methodology provides a user-friendly framework to represent complex interactions and coordination among robots, thereby addressing the challenges associated with programming individual behaviours and orchestrating their collaborative efforts. Our proposal, again, is guided by ROS framework, which serves as reference for programming robotic systems. By demonstrating the applicability of our methodology in a smart agriculture scenario, we showcase its effectiveness in streamlining the modeling process and facilitating seamless implementation across diverse MRS applications.

6.1 Disciplined Use of BPMN

In this section, we present our approach for modeling the collective behavior of a MRS in ROS using BPMN. We first introduce a subset of BPMN elements we selected for describing the robots' mission. Then, we present a list of guidelines driving the use of the selected BPMN elements to compose a collaboration diagram that specifies actions and interactions of each robot in a MRS.

6.1.1 Selection of BPMN elements for MRSs

The subset of BPMN elements adopted in our approach are illustrated in Figures 6.2 and 6.1. The selected items resulted from extensive discussions aimed at determining the best of over 85 elements to suit the needs of MRSs. The debate was driven by a top-down approach based on the modeling activity

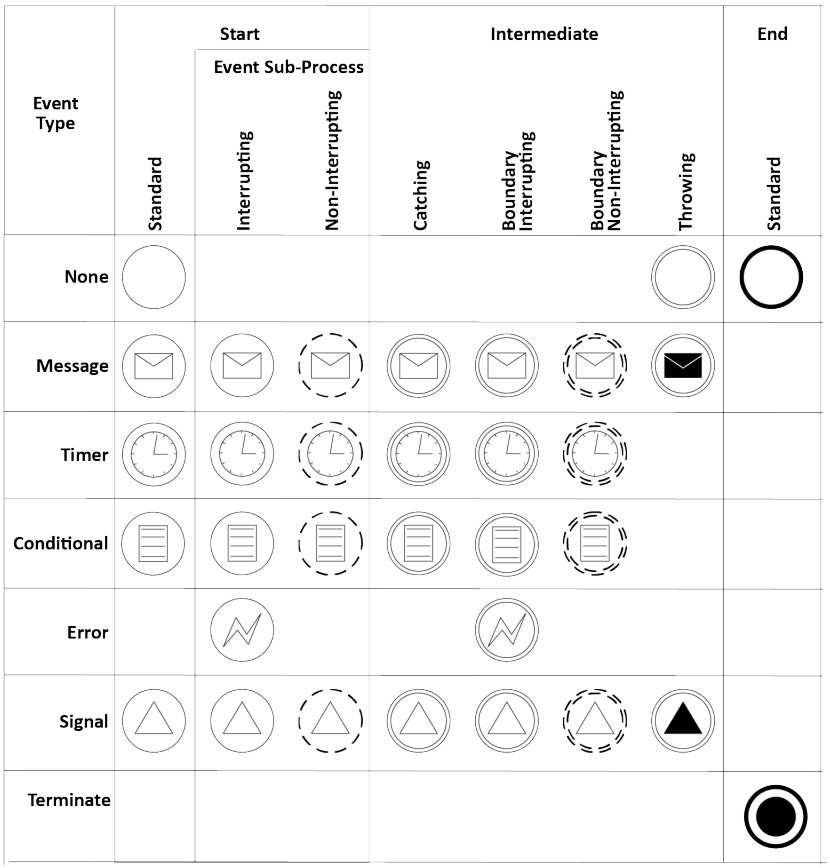


Figure 6.1: Selected BPMN events.

conducted on different application scenarios, and a bottom-up approach informed by experiments carried out with ROS implementations via the Gazebo simulator.

Let us briefly recall the meaning of the BPMN elements resulting from our selection (we refer to Section 2.2 for an introduction to the BPMN notation). A collaboration diagram consists of a collection of (possibly multi-instance) *pools*, which contain processes. In turn, a *process* consists of activities, gateways and events connected to each other through sequence flows, as well as data objects connected to activities and events through data associations. Activities represent one or more pieces of work to be performed within a process. Specifically, a *Call Activity* refers to a call to another process, enabling the structuring of (possibly large and complex) models in terms of decoupled reusable processes. A *Script Task* refers to a piece of code to be executed. An *Event Sub-Process* is a sub-process that is not part of the normal flow of its parent process. *Gateways* are used to control the execution flow of a process, managing parallel activities (AND gateway) and internal/external choices (XOR and Event-based gateways). *Events* represent something that can happen at the beginning, during, or at the close of the process execu-

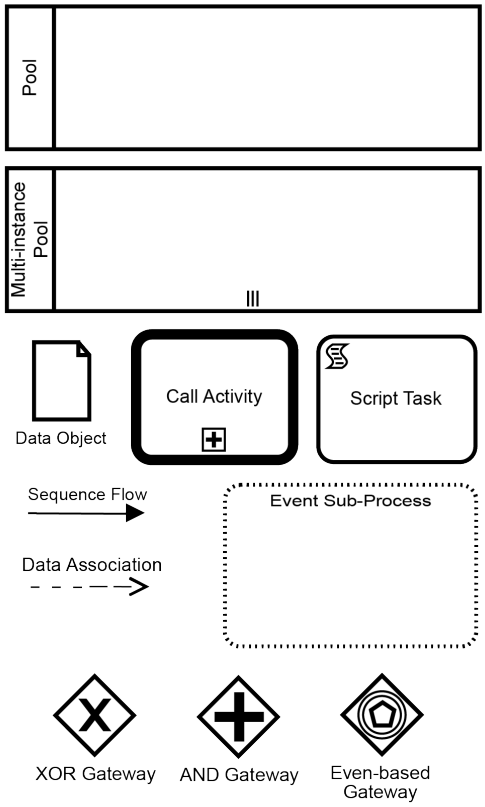


Figure 6.2: Selected BPMN elements.

tion (*Start*, *Intermediate*, or *End* events, respectively). *Start* events serve as alternative entry points for enacting a process. When used in an event sub-process, they can interrupt or not the parent process. *Intermediate* events can happen during the normal flow of the process, by connecting them with sequence flows, or during an activity execution, by attaching them to the activity border. *End* events represent the possible termination of a process or a sub-process. All these events can be further characterized to describe their type and a different semantic meaning. Specifically, we consider *Timer* and *Conditional* events, to react respectively on a time delay or a condition; *Error* events, to throw or catch execution errors; *Signal* events, to describe broadcast and point-to-point communications that may carry data (see [13, p. 235]); and the *Terminate* events to kill all the processes in a pool. Finally, *Data objects* represent information and material flowing in and out of activities and events.

6.1.2 Example Scenario

To better reason on how to specify the behavior of a MRS, we present here an example depicting a smart agriculture scenario.

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

6.1.3 Guidelines for MRS modeling.

We provide a list of guidelines for modeling an MRS through a BPMN collaboration. These guidelines define a disciplined use of the BPMN elements introduced earlier to represent an MRS's cooperative behavior while leaving the designer enough freedom to specify almost any MRS mission. Considering the running scenario of Section 6.1.2, the BPMN collaboration in Figure 6.3 is a possible result of our approach; we refer to it to better present the following guidelines.

G1 Robots as pools. Robots involved in a MRS are abstracted by pools, representing the participants in the collaboration.

G1.1 Heterogeneous robots as single-instance pools. Robots that are heterogeneous, i.e., robots of a different kind or robots with different missions, are abstracted by single-instance pools. Considering Figure 6.3, the drone is represented as a single-instance pool.

G1.2 Homogeneous robots as multi-instance pools. Robots that are homogeneous, i.e., robots of the same kind with the same mission, are abstracted by multi-instance pools. This simplifies the resulting diagram, as a multi-instance pool represents many robots in terms of several instantiations of the same process, avoiding the repetition of the same robot process into different pools. In Figure 6.3, the tractors are represented as a multi-instance pool.

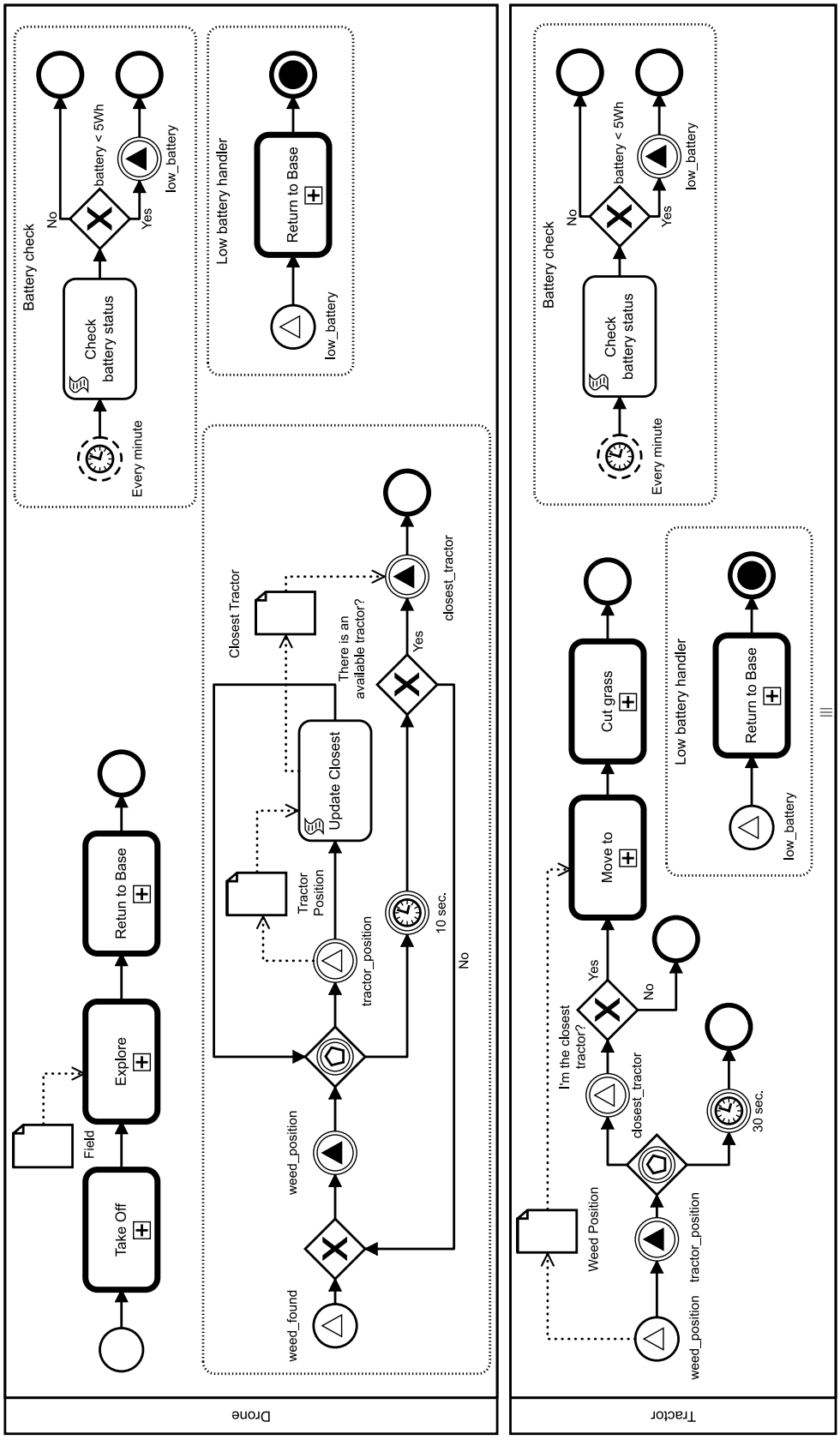


Figure 6.3: Collaboration diagram of the smart agriculture scenario.

G2 Mission as a process. The mission to be performed by a robot is abstracted by a process diagram within the pool of the considered robot. The process diagram expresses the control flow of the mission. For instance, the process contained into the drone pool in Figure 6.3 depicts the robot's behavior from the take-off until the return to the base. Activities are related to each other via sequence flows, to define their execution order, and via gateways (see guidelines G2.3-2.5), to control the execution flow.

G2.1 Actions as activities. A robot mission is mainly made by a set of actions; these actions are abstracted by activities within the mission process diagram. Based on the complexity of the action to specify, we distinguish the type of activity to use in the model between call activities and script tasks.

G2.1.1 Complex actions as call activities. Complex actions, e.g., navigation, perception and control, that can be decomposed into several steps and/or reuse already modeled procedures, are abstracted by call activities. Indeed, a call activity can be used for referencing another process diagram or other existing activities. This enables the modularisation of diagrams and lowers their dimension, speeding up the modeling of the MRS through the re-use of already modeled behaviors. Considering Figure 6.3, the call activity *Return to Base* appears three times in the diagram, indicating that in all these cases the called procedure is implemented in the same way.

G2.1.2 Simple actions as script tasks. Simple actions, which does not require any further decomposition, are abstracted by script tasks. This element refers to a piece of code to be executed by the considered robot. Considering Figure 6.3, to get the tractor closest to the weed, it is used the script task *Update Closest* that performs simple mathematical operations.

G2.2 Event handlers as event sub-processes. Procedures handling an event (such as the expiration of a timer, the satisfaction of a condition, the occurrence of an error, or the reception of a signal) are abstracted by event sub-processes. Indeed, this element triggers an handling process concurrent to the main process describing the robot mission. Based on the event type, the main process can be interrupted or not. For example, checking the battery after an amount of time, as depicted in Figure 6.3, can be performed in parallel with the mission, and thus it is abstracted by a non-interrupting start event.

G2.3 Concurrent behaviors by means of AND gateways. Concurrent behaviors in a robot mission are rendered by means of AND split gateways.

G2.4 Internal choices as XOR gateways. Internal choices in a robot mission are rendered by XOR split gateways. In Figure 6.3, the decision taken on the basis of the battery charge in the *Battery check* event subprocess is rendered as a XOR gateway, with two possible outcomes.

G2.5 External choices as event-based gateways. Choices driven by events in a robot mission are abstracted by means of event-based gateways. Referring to Figure 6.3, a tractor waits for a *closest_tractor* signal at most for 30 seconds by means of an event-based gateway followed by two catching events.

G2.6 Missions activation as start events. The beginning of a mission is abstracted by a start event. In more detail, a none start event fires immediately the robot mission. The other event types activate the mission when a specific circumstance occurs. Referring to Figure 6.3, the main process of the drone contains a none start event triggered at the system start-up, while the tractors are activated only when they receive the *weed_position* signal.

G2.7 Missions shutdown as end events. The end of a mission is abstracted by an end event. The none end event stops only the current execution flow of a mission, while the terminate end event completely stops the mission.

G3 Communication via signal events. Intra- and inter-robot communications, even in presence of a payload, are abstracted by signal events. The sending of a message corresponds to a throwing signal event, and the receiving of a message is made by a catching signal event. The correlation between one or more senders and one or more receivers is made by means of the event name. Considering Figure 6.3, the signal event *weed_position* corresponds to the same-named topic of which the drone is the only publisher, and the tractors are the subscribers.

G4 Data as data objects. The data used along the execution of the robot's mission are abstracted by data objects. They provide storage in which activities and signal events can read or write information. We explicitly represent in the model, in terms of data objects, only the information used to drive the decision making and the data exchange in the mission execution. Of course, at implementation level, other data will be required. However, since they are confined within low-level pieces of code and do not play any role at the abstraction level of the model, they are omitted. This permits reducing the complexity of the diagram.

6.2 Related works

In this section, we shed light in the existing works that focus on modeling workflows of robotic activities. Bozhinoski et al. [101] describe a tool for the mission specification for a team of multicopters and the generation of a detailed flight plan, by a custom DSL that is translated into an intermediate language which represents the basic actions of an aerial vehicle. Exploiting the same tool, Ciccozzi et al. [80] define a set of DSLs to specify the civilian missions for unmanned aerial vehicles. The mission specification is done through a web-based graphical interface, that communicates with some ROS-based controllers which send commands to the copters. A use of the BPMN notation is shown by De la Croix et. al [102], which describe the TRACE tool to tailor BPMN to the robotics domain in order to model a sequence of robotic activities. The aim of this approach is to understand what will happen after an unplanned event and check if it will compromise the mission. The authors applied their proposal to a single robot equipped with ROS, which can execute its tasks and autonomously act to unexpected events. Otsu et al. [103] present an application of the TRACE tool to a multi-robot scenario. The mission is specified in a BPMN file, uploaded inside all the vehicles, and each block of the model corresponds to a robot behavior. The system uses the ROS framework with an open-source package that allows a custom message passing among multiple master nodes. This approach could be avoided and improved by using ROS2 with DDS. Another integration of BPMN standard with an autonomous robot is presented by Rey et al. [104]. The authors automatize a warehouse process by implementing a human-robot cooperation system managed by a web interface able to control the entire process in real-time.

Chapter 7

From BPMN to X-Klaim: A Systematic Methodology for Model Translation and Program Generation

This chapter outlines the translation of BPMN collaboration diagrams into X-Klaim programs for MRSs. BPMN diagrams enhance the readability and usability of MRS designs, while X-Klaim enables process control over robot interactions. We detail a comprehensive mapping from BPMN to X-Klaim constructs, ensuring seamless transition from design to execution. Through this integration, we aim to improve MRS programming efficiency, increasing its accessibility and paving the way for direct execution.

7.1 The Process of Translation

The process of translating a BPMN model into a X-Klaim code, a vital phase of our systematic methodology, is designated to be user-friendly. Although it is based on an in-depth understanding of the structural and semantic nuances of both languages and a recognition of complexities of multi-robot systems, the users are not required to delve into these intricacies. At the core of this translation process lies the mapping of each BPMN element to its equivalent X-Klaim constructs. This alignment 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.

To ensure a smooth translation process, we adhere to set a guiding principles. First, we emphasize the maintenance of system integrity. This means that the translated X-Klaim program should reflect the same behavior as

the original BPMN model. Second, we strive for accuracy and efficiency in the resulting X-KLAIM code. This ensures that the multi-robot system will function as expected without unnecessary computational overhead. We aim for readability and maintainability in the X-KLAIM program. 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 section. 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.

7.1.1 Mapping of BPMN elements to X-KLAIM constructs

For successful translation of BPMN elements into X-KLAIM, the diagrams must be well-structured and adhere to the guidelines presented in the formal classification of BPMN collaborations [47]. A brief explanation of what constitutes a well-structured diagram can be found in subsection 2.2.4 of section 2.2. The working principles of X-KLAIM necessitates each BPMN element to dispatch a tuple representing a token marking an outgoing edge, before transitioning to the next element. Consumption of this tuple is essential before the transition, ensuring process synchronization and execution order, thereby eliminates the possibility of conflicts or inconsistencies in the final output.

BPMN Events. In X-KLAIM, both messages and signals are represented as tuples. For messages, these values embody the transmitted data, while for signals, the tuple values typically indicate an event or state change. We refer to Tables 7.1, 7.2 and 7.3 for a comprehensive mapping of BPMN events to their X-KLAIM equivalents.

Unicast and Multicast Communications. In the context of X-KLAIM, effective management of unicast and multicast communication scenarios significantly aids in the seamless operation of distributed processes. Unicast communication is characterized by Message Throw/-Catch Event within BPMN. In X-KLAIM, unicast communication is enacted through a pair of operations: an outgoing message, expressed as `out(message)@receiverLoc`, originating from the current process and directed to a specific recipient, followed by the receipt of the message at the recipient's location, expressed as `in(message)@self`. This operation sequence ensures that the one-to-one communication setup inherent in unicast scenarios is accurately maintained, mimicking the precise message

passing semantics of BPMN. For multicast communication, on the other hand, it operates on a one-to-many interaction model, paralleling the structure of a Signal Throw/Catch Event within BPMN. X-KLAIM represents multicast communication through a distinct pair of operations: the emission of a signal from the current process into the tuple space, represented as `out(signal)@self`, followed by the reception of this signal by multiple recipients, represented as `read(signal)@senderLoc`. Intriguingly, 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 multicast communication. The tuple representing the signal is then removed when a given timeout (specified by the programmer via the constant Signal_Duration) expires. This operation sequence accurately mirrors the multicast communication semantics of BPMN. For scenarios involving non-interrupting events, X-KLAIM extends its capabilities with `read_nb` and `in_nb` constructs, which allow for non-blocking `read` and `in` operations, respectively. These non-blocking operations permit processes to continue operation even if the incoming message or signal has not been fully consumed. This robust handling of non-interrupting events is an essential aspect of translation from BPMN to X-KLAIM, reinforcing the faithfulness of the translation process.

BPMN Gateways. 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. Conversely, the Parellel Gateway initiates simultaneous execution of multiple paths, demanding the completion of all paths before the gateway can exit. Moreover, X-KLAIM enables the translation of loops, which allow repeated execution of a BPMN element based on a while loop, see Table 7.4.

BPMN Activities. The BPMN elements, such as Call Activity, Event Sub-Process are translated into a new process that is defined separately and given a unique name. This process is evaluated with an activity that has an initialization of code and an output edge, which is used to indicate the completion of the activity. Similarly, Script Task is translated into an activity with an initialization code and an output edge, see Table 7.5.

BPMN Flow. The sequence flow construct is translated by ensuring that the action of the first BPMN element($P1$) is finished before proceeding with the next BPMN element($P2$), by consuming the tuple that corresponds to the edge connecting the two elements, see Table 7.6.

BPMN Pool. As depicted in Table 7.7, The translation of Pool and Multi-Instance Pool in X-KLAIM involve creating a collaboration network with participant nodes that contain the translated processes. The Pool translation is straightforward, as it only requires the definition of the collabora-

tion name and physical address, with the participant node containing the translated process. However, for Multi-Instance Pool translation, multiple participant nodes need to be defined, each containing the translated process. The collaboration name and physical address are also required for this translation. These translations highlight the ability of X-KLAIM to handle complex collaborative processes involving multiple participants.

BPMN Data Object. The Data Object construct is mapped to tuple that have multiple attributes or properties, and allow for easy passing multiple data elements as single parameter in messages between processes. Simple data also can be mapped to a tuple in X-KLAIM, see Table 7.8.

It is important to note that the mapping from BPMN to X-KLAIM follows a compositional approach. This means that each BPMN construct is translated independently and then combined together 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.

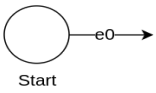
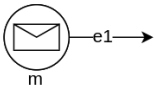
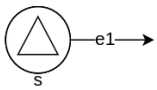
BPMN Element	X-KLAIM Translation	Note
None Start Event 	**out**(e0)**@self**	• Ensure the correct edge-tuple mapping.
Message Start Event 	**in**(m)**@self** **out**(e1)**@self**	• Message flows are required. • Ensure the message is correctly mapped to the tuple. • Ensure the correct edge-tuple mapping.
Signal Start Event 	**read**(s)@senderLoc **out**(e1)**@self**	• Ensure the signal is correctly mapped to the tuple. • Require sender location to be specified corresponding to the process sending the signal. • Ensure the correct edge-tuple mapping.

Table 7.1: Start Events

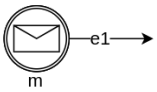
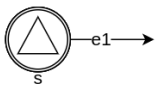
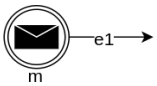
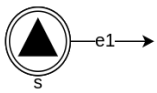
BPMN Element	X-KLAIM Translation	Note
Message Intermediate Catch Event 	**in**(m)**@self** **out**(e1)**@self**	• Similar to Message Start Event • For non-interrupting `in_nb` is used instead.
Signal Intermediate Catch Event 	**read**(s)@senderLoc **out**(e1)**@self**	• Similar to Signal Start Event • For non-interrupting `read_nb` is used instead.
Message Intermediate Throw Event 	**out**(m)@receiverLoc **out**(e1)**@self**	• Message flows are required. • The receiver's location can be determined based on the destination of the message flow. • Ensure the message is correctly mapped to the tuples. • Ensure the correct edge-tuple mapping.
Signal Intermediate Throw Event 	**out**(s)**@self** Thread.sleep(Signal_Duration) **in**(s)**@self** **out**(e1)**@self**	• Ensure the signal is correctly mapped to the tuple. • Ensure the correct edge-tuple mapping. • Signal_Duration is a constant with a timeout value in milliseconds.

Table 7.2: Intermediate Events

BPMN Element	X-KLAIM Translation	Note
None End Event 	stop()	
Message End Event 	**out**(m)@receiverLoc stop()	• Message flows are required. • The receiver's location can be determined based on the destination of the message flow. • Ensure the message is correctly mapped to the tuples. • Ensure the correct edge-tuple mapping.
Signal End Event 	**out**(s)**@self** Thread.sleep(Signal_Duration) **in**(s)**@self** stop()	• Ensure the signal is correctly mapped to the tuples. • Ensure the correct edge-tuple mapping. • Signal_Duration is a constant with a timeout value in milliseconds.

Table 7.3: End Events

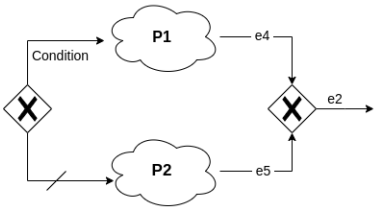
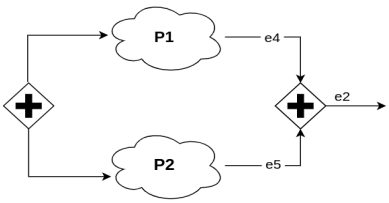
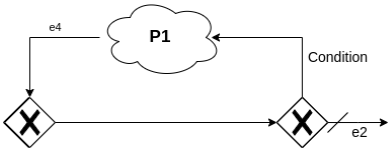
BPMN Element	X-KLAIM Translation	Note
XOR Gateway 	**if**(condition){ translate(P1) **in**(e4)**@self** } **else**{ translate(P2) **in**(e5)**@self** } **out**(e2)**@self**	• Ensure the correct use of `in` and `out` operations with appropriate edge identifiers. • Gateways should be properly connected to their corresponding processes using well-defined edges. • The order of execution enforced by the gateways should correspond accurately with the BPMN model. • Ensure the correct edge-tuple mapping in conditional statements.
AND Gateway 	translate(P1) translate(P2) **in**(e4)**@self** **in**(e5)**@self** **out**(e2)**@self**	• Similar to XOR Gateway.
Loop 	**while**(condition){ translate(P1) **in**(e4)**@self**} **out**(e2)**@self**	• Similar to XOR Gateway.

Table 7.4: BPMN Gateways

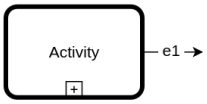
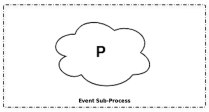
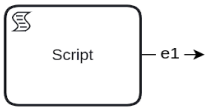
BPMN Element	X-KLAIM Translation	Note
Call Activity 	**eval**(**new** Activity(e1))**@self** *// Process to be added to the node* **proc** Activity(String edgeOut){ { ... initialization code ... } **out**(edgeOut)**@self** }	• Make sure that each activity is properly defined and initialized. • Ensure that the activity is correctly translated into the corresponding process and given a unique name.
Event Sub-Process 	**eval**(**new** EventSubProcess())**@self** *// Process to be added to the node* **proc** EventSubProcess(){ translate(P) }	• Similar to Call Activity.
Script Task 	*// Snippet code of the task* { ... initialization code ... } **out**(e1)**@self**	

Table 7.5: BPMN Activities

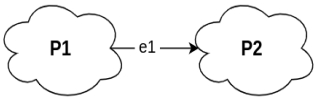
BPMN Element	X-KLAIM Translation	Note
Sequence 	**in**(e1)**@self**	• Use the `in` operation to ensure that the action of the first process is finished before starting the next one. • Ensure the tuple corresponds the edge connecting two Elements (P1) and (P2).

Table 7.6: Sequence Flow

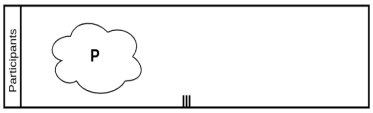
BPMN Element	X-KLAIM Translation	Note
Pool 	**net** Coll_name **physical** "localhost:9999" { **node** participant { translate(P) } }	• Ensure the correct definition of the collaboration network with participant nodes.
Multi Instance Pool 	**net** Coll_name **physical** "localhost:9999" { **node** participant1 { translate(P) } . . **node** participantN { translate(P) } }	• Ensure the correct definition of the collaboration network with participant nodes. • Ensure the correct creation of multiple particpant nodes, each containing the translated process. • N is the maximum number of instances of the pool (specified as an attribute of the pool element).

Table 7.7: BPMN Collaboration

BPMN Element	X-KLAIM Translation	Note
Data 	("Data", datatype attribute1, datatype attribute2, ...)	Make sure the data objects are correctly represented as tuples with correct attributes.

Table 7.8: Data Flow

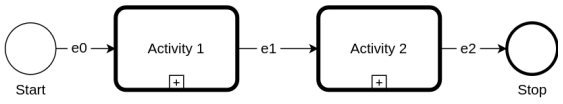
7.1.2 Examples of BPMN processes translated into X-KLAIM

In the following section, we provide examples of BPMN processes and their translations to X-KLAIM.

Example 1

The BPMN process in Figure 7.1, consisting of a start event, two call activities, and an end event. In order to translate the process into X-KLAIM, the sequences of elements in the BPMN graph are mapped to the corresponding X-KLAIM code. We begin with the start event, which is translated into an equivalent X-KLAIM construct. Following this, the sequence connecting the

start event to the first call activity is translated into X-KLAIM. This pattern of sequential translation continues for the remaining elements in the diagram until we reach and translate the final end event.

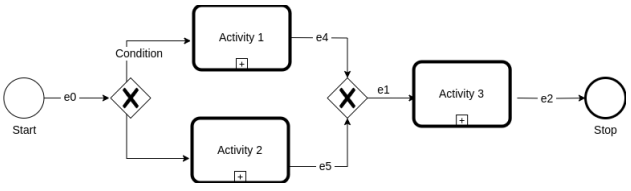



```
out(e0)@self
in(e0)@self

eval(new (Activity 1(e1)))@self
in(e1)@self

eval(new (Activity 2(e2)))@self
in(e2)@self

stop()
```

Figure 7.1: The BPMN graph and its X-KLAIM translation

Example 2

Consider the BPMN process shown in Figure 7.2, comprising a start event, an XOR gateway to choose between two call activities, a call activity, and an end event. The translation of this process into X-KLAIM follows a systematic process. We initiate the translation with the start event, moving it into its equivalent X-KLAIM form. The sequence connecting the start event to the XOR gateway is then mapped into X-KLAIM. Next, the XOR gateway, inclusive of the elements within it, is translated, and the sequence leading to the subsequent element is also converted. The final call activity is then translated into X-KLAIM, as in the sequence linking it to the end event. To conclude, the end event is translated. This careful step-by-step yields a faithful and reliable X-KLAIM representation of the original BPMN process, as depicted in Figure 7.2.

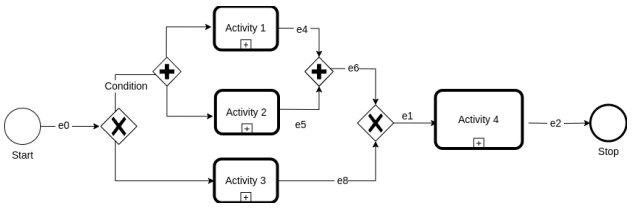



```
out(e0)@self
in(e0)@self

if(Condition){
    eval(new (Activity1(e4)))@self
    in(e4)@self
}
else{
    eval(new (Activity2(e5)))@self
    in(e5)@self
}

out(e1)@self
in(e1)@self


eval(new (Activity3(e2)))@self
in(e2)@self

stop()
```

Figure 7.2: The BPMN graph and its X-KLAIM translation

Example 3

Let us consider a rather complex BPMN graph, depicted in Figure 7.3, that encompasses a start event, an XOR gateway that governs the execution of either two parallel activities inside an AND gateway or a single call activity, and an end event. The translation of this graph into X-KLAIM involves several systematic steps. We initiate the translation with the start event, followed by the mapping of the sequence leading to the XOR gateway. Next, the XOR gateway is converted into X-KLAIM. Depending on the specific condition, the process will either continue to translate the parallel call activities within the AND gateway or translate a singular call activity. After the call activities are transformed into their X-KLAIM equivalents, we move into the sequence leading to the subsequent call activity. Finally, the sequence connecting to the end event is translated, with the translation process culminating with the conversion of the end event. This methodical procedure results in an accurate X-KLAIM representation of the original BPMN process, as demonstrated in Figure 7.3.



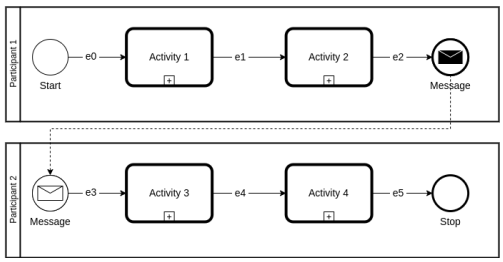

```
out(e0)@self
in(e0)@self

if(Condition){
    eval(new (Activity1(e4)))@self
    eval(new (Activity2(e5)))@self
    in(e4)@self
    in(e5)@self

    out(e6)@self
    in(e6)@self
}
else{
    eval(new (Activity3(e8)))@self
    in(e5)@self
}

out(e1)@self
in(e1)@self

eval(new (Activity4(e4)))@self
in(e2)@self

stop()
```

Figure 7.3: The BPMN graph and its X-KLAIM translation

Example 4

Let us examine a BPMN collaboration diagram, depicted in Figure 7.4, representing a business process involving a two participants. The process for the first participant comprises a start event, two call activities, and a message end event. For the second participant, the process involves a message start event, two call activities, and an end event. The transformation of this BPMN diagram into X-KLAIM is realized in three main stages. The initial stage involves creating a new X-KLAIM network that corresponds to the pool in the collaboration diagram. The network location, or the physical attribute,

is set as `localhost:9999`. The node attributes, which represent the participants in the pool, are also defined at this phase. In the second stage, we focus on translating the internal processes of the first participant. Keeping consistent with our approach in previous examples, the translation begins with the start event, move onto the call activities, and finally ends with the message end event. The third and final stage of the transformation process emphasizes the translation of the internal process of the second participant. This procedure initiates with the message start event, continues with the translation of the two call activities and concludes with the end event. This systematic approach results in an accurate X-KLAIM representation of the BPMN collaboration diagram as shown in Figure 7.4.




```
net Coll name physical "localhost:9999" {
   node participant1 {
      out(e0)@self
      in(e0)@self

      eval(new (Activity 1(e1)))@self
      in(e1)@self

      eval(new (Activity 2(e2)))@self
      in(e2)@self

      out(message)@participant2
      stop()
   }
  node participant2 {
      in(message)@self
      out(e3)@self
      in(e3)@self

      eval(new (Activity 3(e4)))@self
      in(e4)@self

      eval(new (Activity 4(e5)))@self
      in(e5)@self

      stop()
   }
}
```

Figure 7.4: The BPMN graph and its X-KLAIM translation

7.1.3 Code Optimization

A significant facet of this translation process is the focus on code optimization in the X-KLAIM representation. While mapping BPMN elements to X-KLAIM constructs, there is an opportunity to enhance the computational efficiency of the resultant code. In some instances, the process creates sequences that are functionally redundant and can be eliminated without altering the system's behavior. For instance, a sequence where an `out` operation is immediately followed by an `in` using the same tuple merely adds unnecessary steps and computational overhead. Recognizing and eliminating such redundancies improves not only the code's execution efficiency but also its readability and maintainability. Therefore, this optimization step serves a dual purpose: streamlining code execution and enhancing its overall quality.

7.1.4 Prototype of BPMN2XKLAIM Tool

As part of this thesis, we have developed an initial prototype for a tool, named BPMN2XKLAIM. This tool is designed to facilitate the generation of X-KLAIM code from BPMN models. The BPMN models are designed using the Camunda plugin for Intellij. This plugin provides a convenient and intuitive interface for creating BPMN diagrams, which can then be exported as XML files. These '.bpmn' XML files are then parsed by our `BpmnParser`, resulting in the creation of `BpmnElements`. These elements are subsequently fed into our code Generator which translates the BPMN process into X-KLAIM code. To further enhance the efficiency of the generated code, an `Optimizer` function is utilized, refining the code to its most efficient form.

Figure 7.5 provides a visual representation of this translation process. It 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. Below, you can see the corresponding X-KLAIM code produced by our tool from the given BPMN model.

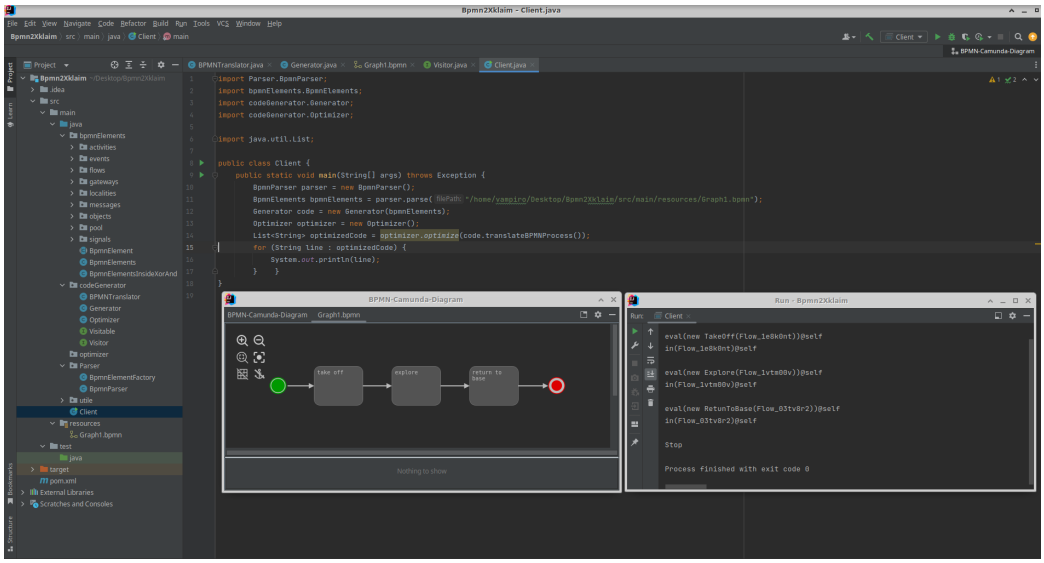


Figure 7.5: BPMN2XKLAIM Tool.

At this stage, the BPMN2XKLAIM tool successfully translates activities, events, XOR gateways, and sequence flows from BPMN diagrams into X-KLAIM code. However, it is currently limited by its inability to handle translation of collaborations, message flows, loop structures, and AND gateways. These elements represent essential components of more complex MRS mission scenarios and their incorporation is necessary for the tool to truly reflect the diversity and intricacy of real-world missions.

Despite these limitations, the BPMN2XKLAIM tool, even in its initial version, demonstrates considerable potential. It has made significant strides in automating the transformation process from BPMN models into X-KLAIM code, contributing substantially to the simplification of this procedure.

7.2 Translation of the Agriculture Scenario

This section provides a comprehensive discussion of the translation of the agriculture scenario from BPMN to X-KLAIM, which was initially presented in Chapter 6. The original BPMN model was modified to be well-structured, adhering to the requirements of the systematic mapping methodology discussed previously. Specifically, the use of event-based gateway was omitted as the mapping of this particular element to X-KLAIM has not been addressed in this thesis. We will begin with a discussion on the translation of the collaboration aspect, detailing how the interactions and dependencies between the Drone and Tractors are captured. We then proceed to the translation of the individual missions of the Drone and Tractors, illuminating how each task within their respective main mission is represented in X-KLAIM. Lastly, we will illustrate how event subprocesses, crucial to the responsiveness and adaptability of the system, are mapped to X-KLAIM process.

7.2.1 Translation of the collaboration

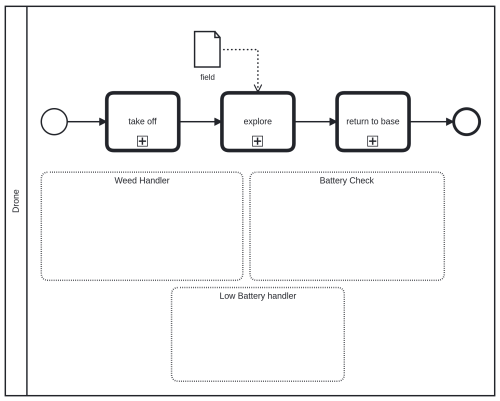
The collaboration within the agriculture scenario involves interactions between the Drone and two Tractors. In X-KLAIM, this is represented as individual nodes for Drone, Tractor1, Tractor2, each executing their respective behavior. In Figure 7.6, we show the graphical representation of the collaboration in BPMN and the corresponding X-KLAIM code.


```
net MRS physical "localhost:9999" {
    node Drone {
        eval(new DroneBehavior(Tractor1,
    Tractor2))@self
    }

    node Tractor1 {
        val robotId = "robot1"
        eval(new TractorBehavior(Drone))@self
    }

    node Tractor2 {
        val robotId = "robot2"
        eval(new TractorBehavior(Drone))@self
    }
}
```

Figure 7.6: Translation of the Collaboration

7.2.2 Translation of the Drone mission

The translation of the Drone mission encapsulates the sequence of operations that the drone undertakes in the agriculture scenario. The Drone's mission is structured around three key tasks: taking off, exploring the field, and returning to the base. In the BPMN representation, this flow of activities is depicted through the use of event and call activities. In the corresponding X-KLAIM translation, each of these activities is represented as a process within the primary `DroneBehavior` process. Additionally, the event subprocesses, which handle battery check, low battery, and weed handling, are initiated concurrently with the main mission. In Figure 7.7, we show the graphical representation of the Drone mission in BPMN and the corresponding X-KLAIM code.




```
proc DroneBehavior(Locality Tractor1,
Locality Tractor2) {

   // Initiate event subprocesses for
      // batery checks,handling of low
      // battery events and weed handling

   eval(new CheckBattery())@self
   eval(new LowBatteryHandler())@self
   eval(new WeedHandler())@self

   // Start the main mission

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

      // Drone explores field after takeoff
   in(e1)@self
   eval(new Explore(field, e2))@self

      // Drone returns to base after
      // exploration
   in(e2)@self
   eval(new ReturnBase(e3))@self

      // Missions ends after returning
      // to base
   in(e3)@self
      Stop()
}
```

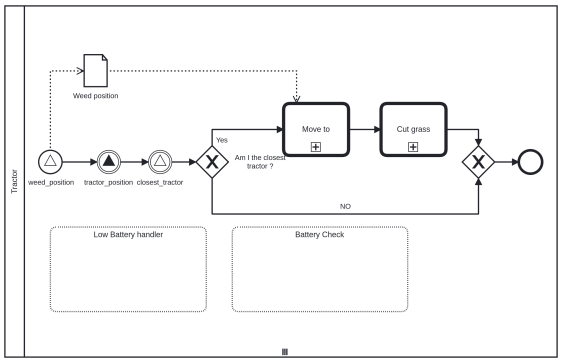
Figure 7.7: Translation of the Drone mission

This translation not only maintains the sequential order of the operations but also represents the concurrent subprocesses that could potentially interrupt the main mission. The aim is to provide a comprehensive representation of the drone's autonomous mission behavior, maintaining the operational sequence while accounting for possible event-drive, interruptions.

7.2.3 Translation of the Tractor mission

The translation of the Tractor mission embodies the tasks performed by the tractors in the agricultural scenario. In this sequence, the Tractor primarily waits for signals from Drone, takes actions based on the information received,

and executes tasks such as moving to a specified location and cutting grass. In the BPMN model, this behavior is represented as a series of signal events, call activities, and conditional gateways. This allows the Tractor's actions to be driven by the signals it receives from the Drone and conditions in the field. In the corresponding X-KLAIM representation, each activity and event is translated into a process within main `TractorBehavior` process. As with Drone, event subprocesses are also present for the Tractor, handling battery checks and low battery events. In Figure 7.8, we show the graphical representation of the Drone mission in BPMN and the corresponding X-KLAIM code.




```
proc TractorBehavior(Locality Drone,
String robotId) {

   // Initiate event subprocesses
   eval(new CheckBattery(robotId))@self
   eval(new LowBatteryHandler(robotId))@self

   // Tractor reads position of the weed
   // from Drone
   read(WEED_POSITION,weedPos)@Drone

   // Tractor sends its position to Drone
   out(TRACTOR_POSITION, tractorPos)@self
   Thread.sleep(Signal_Duration)
   in(TRACTOR_POSITION, tractorPos)@self

   // Tractor reads information about
   // the closest tractor from Drone
   read(CLOSEST_TRACTOR)@Drone

   if(ClosestTractor){
      // If it is the closest tractor,
      // it moves towards the weed position
      eval(new MoveTo(robotId, weedPos,
      e4))@self

      // Tractor cuts the grass after moving
      // to the weed position
      in(e4)@self
      eval(new CutGrass(robotId,e5))@self

      // Tractor mission ends after
      // cutting the grass
      in(e5)@self
      Stop()
    }
    else{
      // Tractor mission ends if it's not
      //the closest tractor
      Stop()
   }
}
```

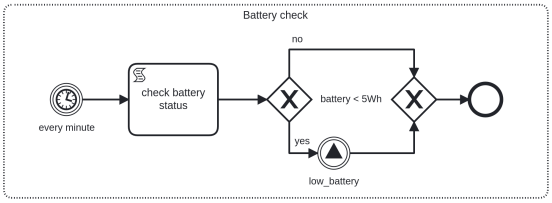
Figure 7.8: Translation of the Tractor mission

The translation in X-KLAIM aims to capture the autonomous behavior of the tractors as they react to changes in the environment, maintaining the order of operations as they are triggered by external signals and conditions. This effectively encapsulates the Tractor's reactive behavior in a manner consistent with the overall collaborative robotic system design.

7.2.4 Translation of event-subprocess

Event sub-processes are vital components of the BPMN models that allow for the handling of events that can occur at any time during the process execution. In our agricultural scenario, such subprocesses are used to monitor and manage battery-related events and the weeds. As a matter of example, we discuss here the translation of the battery check subprocess; the other event-subprocesses are translated similary.

In the BPMN model, the battery check is an event subprocess that continuously monitors the battery levels of the robotic entities. If a low battery status is detected, a signal is sent to initiate the appropriate procedures. The corresponding X-KLAIM code captures this behavior using a recursive process. The `CheckBattery` checks the battery status at a regular intervals. If a critical battery level is detected (i.e, less than 5Wh in our scenario), a low battery signal is emitted. After each check, the process invokes itself recursively, creating a continuous cycle of battery checks. In Figure 7.9, we show the graphical representation of the Battery check subprocess in BPMN and the corresponding X-KLAIM code.




```
proc CheckBattery(String robotId){
   // The timer
   Thread.sleep(60)

   // Implement here the code to
   // check battery status
   {Initizaliation of code}

   // If battery level is cricial
   if(battery < 5Wh) {

     //Emit low battery signal
     out(LOW_BATTERY)@self
     Time.sleep(signal_duration)
     in(LOW_BATTERY)@self

     stop()
   }
   else{

     // No cricial battery status detected
     stop()
   }
   // Repeat the battery check process
   eval(new CheckBattery(robotId))
}
```

Figure 7.9: Translation of Battery check subprocess

7.3 Related works

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 the translation of BPMN diagrams to Petri Nets, thereby enabling the application of existing formal verification mechanism [105], while

others have leveraged Petri Nets to develop solutions for process collaboration [106]. 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 [107]. In parallel, a number of studies have employed Business Process Execution Language (BPEL) as the target language, focusing on transition from BPMN to BPEL to expand the practical execution possibilities for BPMN diagrams [108, 109]. A significant body of research has also utilised 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 [110, 111]. Similarly, the π-calculus has also been adopted as a target language in the mapping [112, 113]. Other noteworthy translations include those to Concurrent Process Modeling (CPM) to facilitate the direct execution of process models [114], to the mathematical Calculus for Orchestration of Web Services (COWS) for orchestrating web services [115], and to Recursive ECATNets, which can be expressed in terms of conditional rewriting logic [116].

Our approach, mapping BPMN to X-KLAIM, distinctly sets itself apart from existing work through several key aspects. Unlike previous studies that 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, make the output code amenable to formal analysis, paving the way for future robustness and safety considerations in complex systems like autonomous robots. Furthermore, our method's compatibility with the prevalent ROS middleware and potential adaptability to its future versions add to its applicability in modern robotics software infrastructures [20, 21].

PART IV

CONCLUSIONS

Chapter 8

Concluding remarks

As we draw our journey through the exploration of heterogeneous robotics applications programming to a close, it's time to reflect upon the research questions we have addressed. Our objective has been to provide a solution that facilitates the development and coordination of MRSs, taking into consideration the complexities of dealing with heterogeneity in both behavior and communication among different robots. We aimed to model and abstract MRS missions at a high level and to effectively bridge the gap between the conceptual design and concrete implementation of MRS missions. Now, let's revise our research questions and discuss how our proposed approach has been able to effectively address each of them.

Addressing RQ1: Programming and Coordinating MRSs

Chapters addressed: Chapter 4 and Chapter 5 (publications [20],[21],[22])

X-KLAIM has proved expressive enough to smoothly implement MRSs' behaviors, and its integration with Java allowed us to seamlessly use the *java_rosbridge* API directly in the X-KLAIM code to access the publish/-subscribe communication infrastructure of ROS. Our experimental results show that the use of X-KLAIM and *java_rosbridge* introduces just a slightly greater but acceptable latency than the traditional ROS implementation based on Python code.

We believe the X-KLAIM computation and communication model is particularly suitable for programming MRSs' behavior. On the one hand, X-KLAIM natively supports concurrent programming, which is required by the distributed nature of robots' software. On the other hand, the organization of an X-KLAIM application in terms of a network of nodes interacting via multiple distributed tuple spaces, where communicating processes are decoupled both in space and time, naturally reflects the distributed structure of an MRS. In addition, X-KLAIM tuples permit to model both raw data

produced by sensors and aggregated information obtained from such data. This allows programmers to specify the robot's behavior at different levels of granularity, thus permitting to structure the code in logical layers that provide a systematic approach to program MRS missions. Moreover, the form of communication offered by tuple spaces, supported by X-KLAIM, favors the scalability of MRSs in terms of the number of components and robots that can be dynamically added and meets the open-endedness requirement (i.e., robots can dynamically enter or leave the system). Both features are crucial in MRSs.

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

Harnessing the expressive nature of X-KLAIM, we build upon our core movement patterns to develop mission scenarios tailored to specific user needs. These patterns, rooted in real-world mission requirements, serve as foundational blocks, allowing us to create, modify, and deploy complex robotic tasks. Through the versatility of these patterns, we have achieved a higher level of flexibility and customization of our MRS mission designs. The adaptability and scalability brought about these patterns have propelled us

towards creating more efficiency and effectiveness.

Moreover, these mission patterns encapsulated in X-Klaim have been vital in fostering better coordination among different processes, concurrent execution of tasks, and facilitation of a distributed approach towards mission planning. By bringing these intricate processes together, we have shown that the most complex of missions can be broken down into simpler, manageable parts - an approach that paves for more ambitious an complex MRS missions in the future.

Addressing RQ2: Modeling and Abstracting MRS Missions

Chapter addressed: Chapter 6 (paper [23])

In our approach of modeling MRS missions at a high level, we identified and utilized a selection of BPMN elements, with specific reference to their implementation within ROS. This selective application ensured that we extracted the most suitable facets of BPMN for MRS missions, aligning closely with the operational intricacies of ROS.

BPMN's unique attributes come into play in its graphical nature, which is adept at representing intricate mission sequences and interactions. This level of abstraction is instrumental in rendering complex mission in an easy-to-understand manner.

The power of BPMN goes beyond representation. The standardized nature of this DSL promotes clarity, precision, and interoperability, resulting in an improved understanding of MRS mission designs. These characteristics reinforce BPMN as an optimal choice for MRS mission modeling.

Finally, the expressiveness of our approach has been validated in complex real-world applications, such as smart agriculture. Our selection of BPMN elements, tailored to MRS missions specifics, proved sufficiently expressive to accommodate the intricacies of such applications. To aid the adoption of our approach, we developed a set of guidelines to streamline the creation of MRS mission models using BPMN. These guidelines ensure consistent and effective application our our approach across various scenarios

Addressing RQ3: Bridging the Gap Between Modeling and Implementation

Chapter addressed: Chapter 7 (paper [24])

Building upon our strides in X-Klaim-based MRS design and our adaptation of BPMN, we then explored a systematic mapping of the BPMN elements to X-Klaim constructs, achieving a high-level of integration between these two tools. This methodology has not only bolstered the clarity and maintainability of our mission designs but has also permitted a more streamlined transition from an abstract BPMN model to a concrete X-Klaim program.

The power of this approach lies in its utility: By starting with BPMN model, roboticists and domain experts can together design missions at a high-level of abstraction, ensuring that the mission's purpose and broad steps are well understood by all stakeholders. Then, the model can be systematically translated into an X-KLAIM program, preserving the structure and logic of the original design while adding the necessary detail for execution in a ROS environment.

This progression - from abstract mission design in BPMN to concrete program implementation in X-KLAIM - is an elegant encapsulation for our research contributions. We have filled effectively the gap between high-level mission planning and low-level mission execution, achieving a balance between abstraction and precision, accessibility and power, design and implementation.

Overall Achievements

Looking back at our journey through these chapters, we appreciate the synergistic fusion of X-KLAIM, BPMN and ROS that we have achieved. Through the use of X-KLAIM for coordinating and programming multiple-ROS based system, we have been able to raise the level of abstraction, enhancing readability and usability. The systematic translation from BPMN to X-KLAIM paves the way for a more integrated and seemless design-to-implementation process.

Chapter 9

Future work

Our long-term goal is to design a domain-specific language for the robotics domain that, besides being used for automatically generating executable code, is integrated with tools supporting formal verification and analysis techniques. These tools are indeed highly desirable for such complex and often safety-critical systems as autonomous robots [117]. The tools already developed for KLAIM, e.g., type systems [118, 119, 120, 121], behavioral equivalences [122], flow logic [123], and model checking [124, 125, 126], could be a valuable starting point. A first attempt to define a formal verification approach for the design of MRSs using the KLAIM stochastic extension StoKlaim and the relative stochastic logic MoSL [125] has been presented in [127]. Along this direction, we plan to investigate the integration of the proposed approach with spatial model checking [128], as done in [129] for a monitoring scenario involving agents moving in physical space. For example, this would permit to guarantee that the robots do not cross unauthorized zones without first signalling themselves in some authorization area, or to verify whether all the items are reachable without crossing a given zone. In addition, as the completion time of the robots' activities may be crucial in some robotics scenarios, we also intend to consider the analysis of spatial-temporal properties, as in [130].

Runtime adaptation is another important capability of MRSs. In [131], we have shown that adaptive behaviors can be smoothly rendered in KLAIM by exploiting tuple-based higher-order communication to exchange code and possibly execute it. We plan to investigate to what extent we can benefit from this mechanism to achieve adaptive behaviors in robotics applications. For example, an X-KLAIM process (a controller or an actuator) could dynamically receive code from other possibly distributed processes containing the logic to continue the execution.

X-KLAIM has several other features that we did not use in this work. We list here the most interesting ones, which could be useful for future work

in the field of MRSs. Non-blocking versions of `in` and `read` are available: `in_nb` and `read_nb`, respectively. These are useful to check the presence of a matching tuple without being blocked indefinitely. Under that respect, X-KLAIM also provides "timed" versions of these operations: as an additional argument, they take a timeout, which specifies how long the process executing such action is willing to wait for a matching tuple. If a matching tuple is not found within the specified timeout, the programmer can adopt adequate countermeasures. In the example of this paper, we used the simplest way of specifying a *flat* and *closed* network in X-KLAIM. However, X-KLAIM also implements the hierarchical version of the KLAIM model as presented in [34], which allows nodes and processes to be dynamically added to existing networks so that modular programming can be achieved and *open-ended* scenarios can be implemented.

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

- By using an existing ROS package that provides AI functionalities. This solution does not require any development effort and is completely transparent to the X-KLAIM code, which can activate and take advantage of the new functionalities by resorting to the publish/subscribe communication mechanism as usual.

- By using existing Python libraries (e.g., TensorFlow[1], Keras[2], PyTorch[3], scikit-learn[4], etc.) to define custom AI models and exposing them as ROS nodes. Again, once the ROS nodes have been created, this solution is completely transparent to the X-KLAIM code.

- By importing an existing Java library (e.g., Deeplearning4j[5], DJL[6], etc.) or a Java wrapper of a library written in another language. This way, the AI functionalities will be directly and easily accessible from the X-KLAIM code, thanks to the interoperability with Java provided by XBASE.

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

Building upon the insights obtained from Chapter 5, we look towards expanding our research in several promising directions. Firstly, we plan to

[1] `https://www.tensorflow.org`
[2] `https://keras.io/`
[3] `https://pytorch.org/`
[4] `https://scikit-learn.org/`
[5] `https://deeplearning4j.konduit.ai/`
[6] `https://djl.ai/`

enhance the functionality of X-KLAIM mission specification patterns by incorporating a broader array of building blocks. This would allow us to more accurately represent a diverse range of multi-robot missions and scenarios, thereby broadening the applicability of our work.

In conjunction with these efforts, we are also working on further developments of our BPMN2XKLAIM tool. The aim is to enhance its capabilities beyond generating X-KLAIM skeleton code from simple BPMN models. This involves refining the tool to support translations of more intricate BPMN elements, such as collaborations, message flows, loop structures, and AND gateways. Our vision is to cultivate a tool that not only reduces potential for human error and increases efficiency, but also can handle a broader range of BPMN elements. By doing so, we intend to make the tool more versatile and capable of dealing with complex multi-robot systems and their missions.

Additionally, we recognize the need to extend our existing BPMN to X-KLAIM mapping to include all selected BPMN elements to MRS modeling presented in Chapter 6. This planned expansion will ensure a more comprehensive and versatile translation capability between the two languages.

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

Lastly, we are keen to validate and expand our work through increased engagement with industry sectors. We plan to conduct further experiments and pilot projects, refining our approaches based on real-world applications and feedback. We believe this will be crucial in ensuring that our work continues to be grounded in practical, industry-relevant considerations.

Bibliography

[1] Dhouib, S., et al. RobotML, a domain-specific language to design, simulate and deploy robotic applications. In *Proc. of SIMPAR*, volume 7628 of *LNCS*, pages 149–160. Springer, 2012.

[2] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In *Proc. of DSLRob'11*, volume abs/1301.7190 of *CoRR*, 2013.

[3] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. A survey on domain-specific modeling and languages in robotics. *Software Engineering for Robotics*, 7:75–99, 2016.

[4] Rajesh Doriya, Siddharth Mishra, and Swati Gupta. A brief survey and analysis of multi-robot communication and coordination. In *Int. Conf. on Computing, Communication, Automation*, pages 1014–1021, 2015.

[5] Rocco De Nicola, Luca Di Stefano, and Omar Inverso. Toward formal models and languages for verifiable multi-robot systems. *Frontiers in Robotics and AI*, 5, 2018.

[6] Quigley, M., et al. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[7] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *SIMPAR*, LNCS 8810, pages 195–206. Springer, 2014.

[8] Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. A survey of Model Driven Engineering in robotics. *Computer Languages*, 62:101021, 2021.

[9] Casalaro, G.L., et al. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Software and Systems Modeling*, 2021.

[10] Zhi Yan, Nicolas Jouandeau, and Arab Ali. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10:1, 12 2013.

[11] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):61–70, 2007.

[12] Brian P Gerkey and Maja J Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International journal of robotics research*, 23(9):939–954, 2004.

[13] OMG. Business Process Model and Notation (BPMN) v. 2.0, 2011.

[14] Compagnucci et al. Modelling notations for IoT-aware business processes: A systematic literature review. In *BP-Meet-IoT*, volume 397 of *LNCS*, pages 108–121. Springer, 2020.

[15] Wei Tan, Yushun Fan, and MengChu Zhou. A petri net-based method for compatibility analysis and composition of web services in business process execution language. *IEEE Transactions on Automation Science and Engineering*, 6(1):94–106, 2008.

[16] Marco Häußler, Sebastian Esser, and André Borrmann. Code compliance checking of railway designs by integrating bim, bpmn and dmn. *Automation in Construction*, 121:103427, 2021.

[17] F. Corradini, C. Muzi, B. Re, L. Rossi, and F. Tiezzi. Formalising and animating multiple instances in BPMN collaborations. *Information Systems*, 101459, 2019.

[18] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Formalising and animating multiple instances in bpmn collaborations. *Information Systems*, 103:101459, 2022.

[19] Corradini et al. A formal approach to modeling and verification of business process collaborations. *SCP*, 166:35–70, 2018.

[20] Lorenzo Bettini, Khalid Bourr, Rosario Pugliese, and Francesco Tiezzi. Writing robotics applications with X-Klaim. In *ISoLA 2020*, volume 12477 of *LNCS*, pages 361–379, Heidelberg, 2020. Springer.

[21] Lorenzo Bettini, Khalid Bourr, Rosario Pugliese, and Francesco Tiezzi. Programming Multi-robot Systems with X-Klaim. In *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning*, volume 13703 of *LNCS*, pages 283–300. Springer, 2022.

[22] Lorenzo Bettini, Khalid Bourr, Rosario Pugliese, and Francesco Tiezzi. Coordinating and programming multiple ros-based systems with X-KLAIM. *International Journal on Software Tools for Technology Transfer*, 2023. to appear.

[23] Khalid Bourr, Flavio Corradini, Sara Pettinari, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Disciplined use of BPMN for mission modeling of multi-robot systems. In *Proceedings of PoEM-Forum*, volume 3045 of *CEUR Workshop Proceedings*, pages 1–10. CEUR-WS.org, 2021.

[24] Khalid Bourr and Francesco Tiezzi. From BPMN to X-KLAIM: A systematic methodology for model translation and program generation. ongoing work, 2023.

[25] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[26] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[27] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[28] L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.

[29] L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *DAIS*, volume 3543 of *LNCS*, pages 181–193. Springer, 2005.

[30] Lorenzo Bettini, Rocco De Nicola, Rosario Pugliese, and Gian Luigi Ferrari. Interactive Mobile Agents in X-Klaim. In *WETICE*, pages 110–117. IEEE Computer Society, 1998.

[31] Lorenzo Bettini, Emanuela Merelli, and Francesco Tiezzi. X-Klaim Is Back. In *Models, Languages, and Tools for Concurrent and Distributed Programming*, volume 11665 of *LNCS*, pages 115–135. Springer, 2019.

[32] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2nd edition, 2016.

[33] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*, pages 112–121. ACM, 2012.

[34] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. An infrastructure language for open nets. In *SAC*, pages 373–377. ACM, 2002.

[35] Rami-Habib Eid-Sabbagh, Remco Dijkman, and Mathias Weske. Business process architecture: use and correctness. In *Business Process Management: 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings 10*, pages 65–81. Springer, 2012.

[36] OMG Omg, R Parida, and S Mahapatra. Business process model and notation (bpmn) version 2.0. *Object Management Group*, 1(4):18, 2011.

[37] Stephen A White et al. Process modeling notations and workflow patterns. *Workflow handbook*, 2004(265-294):12, 2004.

[38] M Weske. Chapter 1: Introduction. *Business Process Management: Concepts, Languages, Architectures. Springer Science & Business Media*, pages 1–24, 2012.

[39] Bruce Silver. *BPMN Method and Style: A Structured Approach Fopr Business Process Modeling and Implementation Using BPMN 2.0*. Cody-Cassidy Press, 2011.

[40] Michele Chinosi and Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[41] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software technology*, 50(12):1281–1294, 2008.

[42] Gero Decker and Alistair Barros. Interaction modeling using bpmn. In *Business Process Management Workshops: BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers 5*, pages 208–219. Springer, 2008.

[43] Michael zur Muehlen and Jan Recker. How much language is enough? theoretical and practical use of the business process modeling notation.

Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE, pages 429–443, 2013.

[44] Gustav Aagesen and John Krogstie. Analysis and design of business processes using bpmn. *Handbook on Business Process Management 1: Introduction, Methods, and Information Systems*, pages 213–235, 2010.

[45] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. *Information Systems*, 37(6):518–538, 2012.

[46] Bartek Kiepuszewski, Arthur Harry Maria Ter Hofstede, and Christoph J Bussler. On structured workflow modelling. In *Advanced Information Systems Engineering: 12th International Conference, CAiSE 2000 Stockholm, Sweden, June 5–9, 2000 Proceedings 12*, pages 431–445. Springer, 2000.

[47] Flavio Corradini, Andrea Morichetta, Chiara Muzi, Barbara Re, and Francesco Tiezzi. Well-structuredness, safeness and soundness: a formal classification of bpmn collaborations. *Journal of Logical and Algebraic Methods in Programming*, 119:100630, 2021.

[48] Bruno Siciliano, Oussama Khatib, and Torsten Kröger. *Springer handbook of robotics*, volume 200. Springer, 2008.

[49] Gregory Dudek, Michael RM Jenkin, Evangelos Milios, and David Wilkes. A taxonomy for multi-agent robotics. *Autonomous Robots*, 3:375–397, 1996.

[50] Lynne E Parker. Alliance: An architecture for fault tolerant multirobot cooperation. *IEEE transactions on robotics and automation*, 14(2):220–240, 1998.

[51] Hiroaki Yamaguchi. A cooperative hunting behavior by mobile robot troops. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, volume 4, pages 3204–3209. IEEE, 1998.

[52] Gerardo Beni and Jing Wang. Swarm intelligence in cellular robotic systems. In *Robots and biological systems: towards a new bionics?*, pages 703–712. Springer, 1993.

[53] M Ani Hsieh, Anthony Cowley, Vijay Kumar, and Camillo J Taylor. Maintaining network connectivity and performance in robot teams. *Journal of field robotics*, 25(1-2):111–131, 2008.

[54] Maria Valera Espina, Raphael Grech, Deon De Jager, Paolo Remagnino, Luca Iocchi, Luca Marchetti, Daniele Nardi, Dorothy Monekosso, Mircea Nicolescu, and Christopher King. Multi-robot teams for environmental monitoring. *Innovations in Defence Support Systems–3: Intelligent Paradigms in Security*, pages 183–209, 2011.

[55] Kshitij Tiwari and Nak Young Chong. *Multi-robot Exploration for Environmental Monitoring: The Resource Constrained Perspective*. Academic Press, 2019.

[56] Joseph L Baxter, EK Burke, Jonathan M Garibaldi, and Mark Norman. Multi-robot search and rescue: A potential field based approach. *Autonomous robots and agents*, pages 9–16, 2007.

[57] Faiza Gul, Imran Mir, Laith Abualigah, and Putra Sumari. Multi-robot space exploration: An augmented arithmetic approach. *IEEE Access*, 9:107738–107750, 2021.

[58] Lynne E Parker, Daniela Rus, and Gaurav S Sukhatme. Multiple mobile robot systems. *Springer handbook of robotics*, pages 1335–1384, 2016.

[59] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7:1–41, 2013.

[60] Markus Hannebauer, Jan Wendler, Enrico Pagello, Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. Reactivity and deliberation: a survey on multi-robot systems. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems: From RoboCup to Real-World Applications*, pages 9–32. Springer, 2001.

[61] A. Farinelli, L. Iocchi, and D. Nardi. Multirobot systems: a classification focused on coordination. *IEEE Trans. on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(5):2015–2028, 2004.

[62] Tucker Balch and Ronald C Arkin. Behavior-based formation control for multirobot teams. *IEEE transactions on robotics and automation*, 14(6):926–939, 1998.

[63] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[64] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. Ant algorithms and stigmergy. *Future generation computer systems*, 16(8):851–871, 2000.

[65] C Ronald Kube and Eric Bonabeau. Cooperative transport by ants and robots. *Robotics and autonomous systems*, 30(1-2):85–101, 2000.

[66] Iain D Couzin, Jens Krause, Nigel R Franks, and Simon A Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.

[67] Tom Wagner, John Phelps, and Valerie Guralnik. Centralized vs. decentralized coordination: Two application case studies. *An Application Science for Multi-Agent Systems*, pages 41–75, 2004.

[68] Vipin P Veetil. Coordination in centralized and decentralized systems. *Available at SSRN 2600735*, 2017.

[69] Lynne E Parker. Distributed intelligence: Overview of the field and its application in multi-robot systems. In *AAAI fall symposium: regarding the intelligence in distributed intelligent systems*, pages 1–6, 2007.

[70] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System.* " O'Reilly Media, Inc.", 2015.

[71] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: the scel language. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):1–29, 2014.

[72] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19:9–30, 2017.

[73] Gregor B Banusić, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. Pgcd: robot programming and verification with geometry, concurrency, and dynamics. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 57–66, 2019.

[74] Nathan P. Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IROS*, pages 2149–2154. IEEE, 2004.

[75] E Estévez, Alejandro Sánchez García, Javier Gámez García, and Juan Gómez Ortega. Art2ool: a model-driven framework to generate target code for robot handling tasks. *The International Journal of Advanced Manufacturing Technology*, 97(1-4):1195–1207, 2018.

[76] James Harbin, Simos Gerasimou, Nicholas Matragkas, Athanasios Zolotas, and Radu Calinescu. Model-driven simulation-based analysis for multi-robot systems. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 331–341. IEEE, 2021.

[77] Alexander Bubeck, Florian Weisshardt, and Alexander Verl. Bride-a toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik 2014; 41st International Symposium on Robotics*, pages 1–6. VDE, 2014.

[78] Adrian Rutle, Jonas Backer, Kolbein Foldøy, and Robin T. Bye. CommonLang: A DSL for defining robot tasks. In *Proc. of MODELS18 Workshops*, volume 2245 of *CEUR Workshop Proc.*, pages 433–442, 2018.

[79] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. Drona: a framework for safe distributed mobile robotics. In *8th Intern. Conference on Cyber-Physical Systems*, pages 239–248, 2017.

[80] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. Adopting mde for specifying and executing civilian missions of mobile multi-robot systems. *IEEE Access*, 4:6451–6466, 2016.

[81] Davide Brugali and Luca Gherardi. Hyperflex: A model driven toolchain for designing and configuring software control systems for autonomous robots. In *Robot Operating System*, volume 625 of *Studies in Computational Intelligence*, pages 509–534. Springer, 2016.

[82] Ritwika Ghosh, Chiao Hsieh, Sasa Misailovic, and Sayan Mitra. Koord: a language for programming and verifying distributed robotics application. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[83] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomas Bures. High-level mission specification for multiple robots. In *Proceedings of the 12th ACM SIGPLAN international conference on software language engineering*, pages 127–140, 2019.

[84] Miyazawa, A., et al. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.*, 18(5):3097–3149, 2019.

[85] David St-Onge, Vivek Shankar Varadharajan, Guannan Li, Ivan Svogor, and Giovanni Beltrame. ROS and Buzz: consensus-based behaviors for heterogeneous teams. *CoRR*, abs/1710.08843, 2017.

[86] Maksym Figat and Cezary Zieliński. Robotic system specification methodology based on hierarchical petri nets. *IEEE Access*, 8:71617–71627, 2020.

[87] Alonso, D., et al. V^{3}CMM: A 3-view component meta-model for model-driven robotic software development. *Journal of Software Engineering for Robotics*, 1:3–17, 2010.

[88] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The brics component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764, 2013.

[89] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. SafeRobots: A model-driven approach for designing robotic software architectures. In *Proc. of CTS*, pages 131–134. IEEE, 2014.

[90] Pranav Srinivas Kumar, William Emfinger, Gabor Karsai, Dexter Watkins, Benjamin Gasser, and Amrutur Anilkumar. Rosmod: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. *Electronics*, 5(3):53, 2016.

[91] Sorin Adam and Ulrik Pagh Schultz. Towards interactive, incremental programming of ROS nodes. In *Workshop on Domain-Specific Languages and models for Robotic systems*, 2014.

[92] Meng, W., et al. Verified ros-based deployment of platform-independent control systems. In *NASA Formal Methods Symposium*, pages 248–262. Springer, 2015.

[93] Sorin Adam, Morten Larsen, Kjeld Jensen, and Ulrik Pagh Schultz. Rule-based dynamic safety monitoring for mobile robots. *Journal of Software Engineering for Robotics*, 7(1):120–141, 2016.

[94] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*, pages 247–254. Springer, 2014.

[95] Rui Wang, Yong Guan, Houbing Song, Xinxin Li, Xiaojuan Li, Zhiping Shi, and Xiaoyu Song. A formal model-based design method for robotic systems. *IEEE Systems Journal*, 13(1):1096–1107, 2018.

[96] Swaib Dragule, Bart Meyers, and Patrizio Pelliccione. A generated property specification language for resilient multirobot missions. In *SERENE*, volume 10479 of *LNCS*, pages 45–61. Springer, 2017.

[97] Chi Hu, Wei Dong, Yonghui Yang, Hao Shi, and Ge Zhou. Runtime verification on hierarchical properties of ROS-based robot swarms. *IEEE Transactions on Reliability*, 69(2):674–689, 2019.

[98] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. A tuple space for data sharing in robot swarms. *EAI Endorsed Trans. Collab. Comput.*, 2(9):e2, 2016.

[99] Rupak Majumdar, Nobuko Yoshida, and Damien Zufferey. Multiparty motion coordination: from choreographies to robotics programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):134:1–134:30, 2020.

[100] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10):2208–2224, oct 2021.

[101] Bozhinoski et al. FLYAQ: Enabling non-expert users to specify and generate missions of autonomous multicopters. In *ASE*, pages 801–806, 2015.

[102] Jean-Pierre de la Croix and Grace Lim. Event-driven modeling and execution of robotic activities and contingencies in the Europa lander mission concept using BPMN. In *i-SAIRAS*. ESA, 2020.

[103] Otsu et al. Supervised Autonomy for Communication-degraded Subterranean Exploration by a Robot Team. In *AeroConf*, pages 1–9. IEEE, 2020.

[104] Rafael Rey, Marco Corzetto, Jose Antonio Cobano, Luis Merino, and Fernando Caballero. Human-robot co-working system for warehouse automation. In *ETFA*, pages 578–585. IEEE, 2019.

[105] Wenjia Huai, Xudong Liu, and Hailong Sun. Towards trustworthy composite service through business process model verification. In *2010 7th International Conference on Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing*, pages 422–427. IEEE, 2010.

[106] Jorge Roa, Omar Chiotti, and Pablo Villarreal. A verification method for collaborative business processes. In *Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I 9*, pages 293–305. Springer, 2012.

[107] Moe Thandar Wynn, HMW Verbeek, Wil MP van der Aalst, Arthur HM ter Hofstede, and David Edmond. Business process verification–finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[108] Zwikamu Dubani, Ben Soh, and Chris Seeling. A novel design framework for business process modelling in automotive industry. In *2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications*, pages 250–255. IEEE, 2010.

[109] Lin Bai and Jun Wei. A service-oriented business process modeling methodology and implementation. In *2009 International Conference on Interoperability for Enterprise Software and Applications China*, pages 201–205. IEEE, 2009.

[110] Peter YH Wong and Jeremy Gibbons. A process semantics for bpmn. In *Formal Methods and Software Engineering: 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings 10*, pages 355–374. Springer, 2008.

[111] Peter YH Wong and Jeremy Gibbons. Formalisations and applications of bpmn. *Science of Computer Programming*, 76(8):633–650, 2011.

[112] Frank Puhlmann and Mathias Weske. Investigations on soundness regarding lazy activities. *Business Process Management*, 4102:145–160, 2006.

[113] Frank Puhlmann. Soundness verification of business processes specified in the pi-calculus. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS: OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, pages 6–23. Springer, 2007.

[114] Yipeng Ji, Hailong Sun, Xudong Liu, Jin Zeng, and Shangda Bai. A decentralized framework for executing composite services based on bpmn. In *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 332–338. IEEE, 2009.

[115] Davide Prandi, Paola Quaglia, and Nicola Zannone. Formal analysis of bpmn via a translation into cows. In *Coordination Models and Languages: 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings 10*, pages 249–263. Springer, 2008.

[116] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. Specification and verification of complex business processes-a high-level petri net-based approach. In *Business Process Management: 13th International Conference, BPM 2015, Innsbruck, Austria, August 31–September 3, 2015, Proceedings 13*, pages 55–71. Springer, 2015.

[117] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems. *ACM Computing Surveys*, 52:1–41, 2020.

[118] Rocco De Nicola, Gian Luigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theor. Comput. Sci.*, 240(1):215–254, 2000.

[119] Daniele Gorla and Rosario Pugliese. Enforcing Security Policies via Types. In *SPC*, volume 2802 of *LNCS*, pages 86–100. Springer, 2003.

[120] Daniele Gorla and Rosario Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *ICALP*, volume 2719 of *LNCS*, pages 119–132. Springer, 2003.

[121] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Confining data and processes in global computing applications. *Sci. Comput. Program.*, 63(1):57–87, 2006.

[122] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Basic observables for a calculus for global computing. *Inf. Comput.*, 205(10):1491–1525, 2007.

[123] Rocco De Nicola et al. From Flow Logic to static type systems for coordination languages. *Sci. Comput. Program.*, 75(6):376–397, 2010.

[124] Rocco De Nicola and Michele Loreti. A modal logic for mobile agents. *ACM Trans. Comput. Log.*, 5(1):79–128, 2004.

[125] Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.

[126] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. *Sci. Comput. Program.*, 99:24–74, 2015.

[127] Gjondrekaj, E., et al. Towards a formal verification methodology for collective robotic systems. In *ICFEM12*, volume 7635 of *LNCS*, pages 54–70. Springer, 2012.

[128] Gina Belmonte, Vincenzo Ciancia, Diego Latella, and Mieke Massink. VoxLogicA: A spatial model checker for declarative image analysis. In *TACAS 2019*, volume 11427 of *LNCS*, pages 281–298. Springer, 2019.

[129] Davide Basile, Maurice H. ter Beek, and Vincenzo Ciancia. An experimental toolchain for strategy synthesis with spatial properties. In *ISoLA 2022*, volume 13703 of *LNCS*, pages 142–164. Springer, 2022.

[130] Vincenzo Ciancia, Stephen Gilmore, Gianluca Grilletti, Diego Latella, Michele Loreti, and Mieke Massink. Spatio-temporal model checking of vehicular movement in public transport systems. *Int. J. Softw. Tools Technol. Transf.*, 20(3):289–311, 2018.

[131] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. Modeling adaptation with a tuple-based coordination language. In *SAC12*, pages 1522–1527. ACM, 2012.

[132] Tomasz Winiarski, Maciej Węgierek, Dawid Seredyński, Wojciech Dudek, Konrad Banachowicz, and Cezary Zieliński. Earl—embodied agent-based robot control systems modelling language. *Electronics*, 9(2):379, 2020.

[133] Gianluca Bardaro and Matteo Matteucci. Using aadl to model and develop ros-based robotic application. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 204–207. IEEE, 2017.

[134] Gianluca Bardaro, Andrea Semprebon, Agnese Chiatti, and Matteo Matteucci. From models to software through automatic transformations: An aadl to ros end-to-end toolchain. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 580–585. IEEE, 2019.

[135] Yingbing Hua, Stefan Zander, Mirko Bordignon, and Björn Hein. From automationml to ros: A model-driven approach for software engineering of industrial robotics using ontological reasoning. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2016.

[136] Dominik Kirchner, Stefan Niemczyk, and Kurt Geihs. Rosha: A multi-robot self-healing architecture. In *RoboCup 2013: Robot World Cup XVII 17*, pages 304–315. Springer, 2014.

[137] Hamza El Baccouri, Goulven Guillou, and Jean-Philippe Babau. Robotic system testing with amsa framework. In *MoDELS (Workshops)*, pages 316–325, 2018.

[138] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering robotics software architectures with exchangeable model transformations. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 172–179. IEEE, 2017.

[139] Nicola Bezzo, Junkil Park, Andrew King, Peter Gebhard, Radoslav Ivanov, and Insup Lee. Demo abstract: Roslab—a modular programming environment for robotic applications. In *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 214–214. IEEE Computer Society, 2014.

[140] Paul Kilgo, Eugene Syriani, and Monica Anderson. A visual modeling language for rdis and ros nodes using atom 3. In *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3*, pages 125–136. Springer, 2012.

[141] Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, and Uwe Aßmann. A role-based language for collaborative robot applications. In *Leveraging Applications of Formal Methods, Verification, and Validation: International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, pages 1–15. Springer, 2012.

[142] David St-Onge, Vivek Shankar Varadharajan, Ivan Švogor, and Giovanni Beltrame. From design to deployment: decentralized coordination of heterogeneous robotic teams. *Frontiers in Robotics and AI*, 7:51, 2020.

[143] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. A tuple space for data sharing in robot swarms. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, page 287–294, Brussels, BEL, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Appendix A

Appendix

This appendix reports the analysis of the DSLs selected in Chapter 3.

DSL	Formal language	High-level	Sub-domain	DSL Type	Multi-robot	Heterogeneous Robots	Coordination
DeRos [91]		✓	S	Tex.			
Earl [132]		✓	T, M	Gra.			
Aadl-based DSL [133, 134]		✓	P, A	Text., Gra.			
PGCD [73, 99]	✓	✓	P	Tex.	✓	✓ (high)	✓
RSSM [86]	✓		M, A	Gra.	✓	(?)	
AutomationML-based DSL [135]	✓	✓	P, A	Gra.	✓	✓	
ART2ool [75]		✓	P, A, G	Gra.			
BCM [77, 88]		✓	P, G	Gra.			
Model-based [95]	✓		M, A	Gra.			
RosHa [136]		✓	C, S	Tex.	✓	(?)	
HyperFlex [81]		✓	P, A	Gra.			
AMSA [137]		✓	S	Gra.			
MATrans [138]		✓	P, A	Tex.			
ROSMOD [90]		✓	A	Gra.			
ROSLab [139, 92]	✓	✓	S, P, A	Tex.			
(HSL)-RMoM [97]	✓		C, S	Tex.	✓		
ROSRV [94]	✓		S	Tex.			
Atom3 [140]		✓	P	Gra.			
FlyAQ [80, 101]		✓	C, T, H	Gra.	✓	✓ (low)	
Noeix [141]		✓	C, T	Tex.	✓	(?)	✓
Koord [82]	✓	✓	P, C, S	Tex.	✓	✓ (high)	✓
ATLAS [76]		✓	M, C	Tex.	✓	✓ (low)	✓
SCEL [71]	✓	✓	P, C, S	Tex.	✓	✓ (high)	✓
BUZZ [98, 85, 142, 143]	✓	✓	C, P	Tex.	✓	✓ (low)	✓
PROMISE [83]		✓	T, H	Gra., Tex	✓	(?)	
ISPL [72]	✓	✓	S	Tex.	✓	✓ (high)	✓
DRONA [79]	✓	✓	C, S, P	Text.	✓		✓
RobotChart [84]	✓	✓	P, S	Gra., Tex.			
CommonLang [78]		✓	T	Text.			

S: Safety and security T: Task and behavior specification
P: Robot programming G: Manipulation and grasping
H: Human-robot interaction C: Robot coordination and collaboration
A: Robot architecture M: Robot modeling and simulation

Table A.1: SLR analysis

DSL	Decentralized Coordination	Compiler	Code Generated	Deployment Platform	IDE Integration	Tool Support
DeRos [91]		✓	C++	ROS	✓	XTEXT
Earl [132]		✓	C++	ROS, OROCOS		FABRIC
Aadl-based DSL [133, 134]		✓	C++	ROS, OROCOS	✓	OCARINA, OSATE
PGCD [73, 99]	✓	✓	Python	ROS		
RSSM [86]		✓	C++	ROS		HPN
AutomationML-based DSL [135]		✓	C++,Python	ROS		(?))
ART2ool [75]		✓	C++	ROS	✓	ART2ool, EMF, Graphiti, Spray
BCM [77, 88]		✓	C++	OROCOS, ROS	✓	(?)
Model-based [95]		✓	C++	ROS		Uppaal
RosHa [136]	✓		(?)	ROS		ALICA
HyperFlex [81]		✓	XML	OROCOS, ROS		HyperFlex
AMSA [137]		✓	C++	ROS	✓	(?)
MATrans [138]		✓	Python	ROS		MontiArcAutomaton,MATrans
ROSMOD [90]		✓	C++	ROS	✓	WebGME
ROSLab [139, 92]			C++	ROS	✓	ROSLab, ROSGen
(HSL)-RMoM [97]			(?)	ROS		(?)
ROSRV [94]		✓	Python	ROS		ROSRV
Atom3 [140]		✓	Python	ROS	✓	ATOM3
FlyAQ [80, 101]			(?)	ROS	✓	FLYAQ
Noeix [141]			P,A,G	Gra.	(?)	(?)
Koord [82]	✓	✓	Python	ROS	✓	CyPhyHouse Toolchain
ATLAS [76]		✓		ROS		ATLAS
SCEL [71]	✓	✓	JAVA	MOOS-IvP		SCEL
BUZZ [98, 85, 142, 143]	✓	✓	Buzz bytecode	ROS, ArGos	✓	buzzc, buzzacm
PROMISE [83]		✓				
ISPL [72]	✓	✓	C++			MCMAS
DRONA [79]	✓	✓	C	ROS		P, Zing
RobotChart [84]		✓	C++	ROS	✓	Xtext, Siruis
CommonLang [78]		✓			✓	Xtext

Table A.2: SLR analysis