Asynchronous Multiparty Communication and Lowcode, Applications to mobile robots

Mohamed Abdallah Cherif
Mamadou Cire Camara
Yassine Dergaoui
Saikou Yaya Barry

Supervisor: Pascal André

M1 ALMA 2023 - 2024

Abstract

This article explores the integration of Model-Driven Development (MDD) into the design and implementation of mobile robotic systems, focusing particularly on automation and low-code applications. Model-driven engineering (MDE) significantly shortens the development life cycle by simplifying design processes, and ensuring compliance with model properties and automating substantial parts of the coding phase. Through our research, we demonstrate how abstract models can be transformed into executable code efficiently to reduce the gap between conceptual design and operational deployment. We're going to illustrate the use of Unified Modeling Language (UML) in creating comprehensive models that cover structural and functional aspects of systems, which are then used to generate customized software for mobile robots. By employing both forward and reverse engineering practices, our approach not only enhances the development process but also provides a methodological framework for ongoing maintenance. The article presents MDD as a key strategy in simplifying the complexity of modern robotic systems in industrial settings.

1 Introduction

The role of software is increasingly prominent in the industry, whether it's through product connectivity, automated production, or the assistance provided by robots and artificial intelligence in services. Software must not only guarantee quality in terms of reliability and performance, it must also be adaptable and upgradeable to meet new needs and constraints. What's more, software maintenance costs, which have traditionally accounted for 70% of overall costs, are continuing to rise. [8].

Model-driven engineering aims to reduce development time by simplifying processes, verifying model property compliance, and partially automating the coding phase. Reasoning to demonstrate system properties is effectively applied at the model level through specific methods and tools. However, this approach is more challenging to implement at the code level due to the complexity of implementation details and distribution constraints, such as security and communication.

Code generation from models has existed for operational models for a long time, whether through compiling source programming languages or using grammar-based generators for specific languages. Examples include tools like lex-yacc for the C language, as well as parsers for XML or JSON. More recently, tools like XTEXT have also emerged. However, generating code from high-level abstraction models derived from software analysis or design remains primarily a developer's prerogative for software engineering work. Automation becomes more cost-effective than manual development when considering evolutionary maintenance of functional, non-functional, or technical requirements such as hardware changes, for example.[2]

In this article, we explore the gradual transition from abstract heterogeneous models to executable source code to shorten the time between analysis and software production. By "heterogeneous model," we mean a model that encompasses the structural, dynamic, and functional aspects of the modeled system. We assume that general-purpose and expressive languages, such as UML or SysML, are used as modeling languages. [12],General-purpose and expressive languages, such as UML or SysML, are used to describe these models. For example, a UML model can be represented by a class diagram or component diagram for the static part, statecharts for the dynamic aspect, and activity diagrams combined with an action language for computations. The generated code corresponds to a system distributed across multiple devices.[1]

The transition from model to code remains a challenge in terms of automation or assistance. On one hand, code generators from UML editors typically produce basic structures requiring additional development. On the other hand, platform-specific tools are more comprehensive, but they often limit the possible applications.[1]

The structure of the article is as follows: we begin with a contextualised introduction, then discuss the various existing approaches and their limitations in section 2. Next, we present a proposed approach based on a model transformation process in sections 3, 4 and 5, where this approach is detailed. Experiments on the previously studied cases are presented in section 6, followed by discussions in section 7. Finally, we conclude the paper by summarising the main points discussed and highlighting future prospects.

2 Background

The general problem addressed is how to efficiently develop and maintain applications based on heterogeneous abstract models, encompassing the structural, dynamic and functional aspects of the modelled system [1].

Regarding software development in advanced engineering, the process begins with needs analysis and extends to the deployment of executable code. We do not address project management or software deployment aspects (DevOps) here. Among the development process models, the Two-Track Unified Process (2TUP or "Y") is chosen for its approach that highlights design as a central activity, merging an analysis model with technical support, as illustrated in the Figure 1 inspirée de [10].

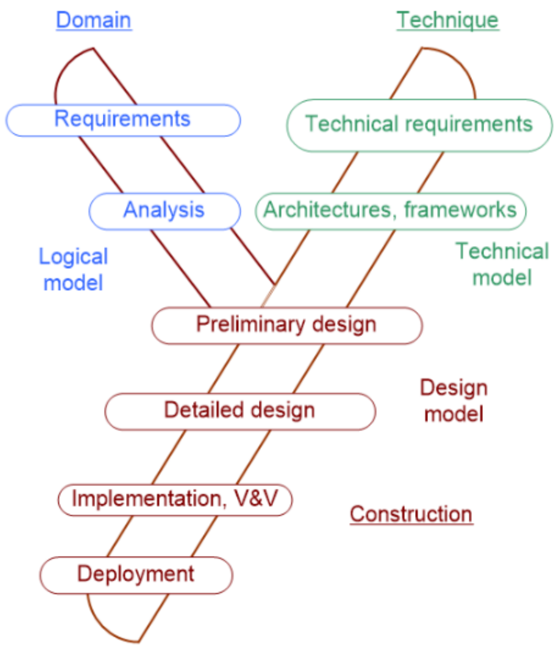


Figure 1: The unified process 2TUP

Since the advent of Model-Driven Architecture (MDA), there has been a belief in the possibility of automatic Model-Driven Development (MDD), promising a reduction in maintenance costs. MDD aims to shorten the development cycle by focusing on abstractions and partially automating code generation. This involves describing abstract models, verifying them, and transforming them to obtain operational applications. Several years later, this goal has not been achieved despite numerous contributions in terms of techniques and tools. Software must not only be of quality in terms of reliability and performance, but it must also be able to evolve and adapt to new needs and constraints [1]. However, to meet these challenges, the community has turned to the automatic generation of code from abstract models. With regard to this approach, a number of UML tools now exist that enable diagrams to be edited. One of their main features is "round-trip engineering", which allows code segments to be defined in the generated code and then reintegrated into the model. In the following we present a few tools and their main functionalities to highlight some important trends in this field [1].

- StarUML is a representative of free UML editors. Each STD is associated with a class representing its protocol. StarUML generates the structural parts of CDs but not the code [2].
- Papyrus is a tool integrable into Eclipse. Papyrus models generation is more advanced than StarUML and allows adding behaviors to operations/functions. However, class relationships were not generated [2].
- Yakindu is a tool specialized in creating statecharts (STD), offered by Itemis. Its main drawback is its ability to process only one STD at a time, i.e., a single class, limiting support for synchronizations between state machines, unless simulating them as parallel sub-regions (composite hierarchical state). Despite this limitation, code generation by Yakindu develops the entire state machine execution logic [2].
- Visual Paradigm is a feature-rich UML modeling tool. It supports generating class diagrams and state-transition diagrams into Java source code as well as C++ or VB.net. It synchronizes code and the model using round-trip engineering functionality [2].
- Modelio is an open-source tool. It distinguishes between the methods it manages and those managed by the developer. A simple method not managed by Modelio is the responsibility of the developer and is retained for round-trip engineering [2].
- IBM Rational Rhapsody is a benchmark tool produced by the designers of OMT, the main inspiration for UML. It is widely used in industries to accelerate the development process, reduce costs, and enhance the quality of embedded software [2].

These listed tools above are not all single-language. For example, Modelio, Papyrus integrate various OMG standards. The MOM (Message Oriented Middleware) line indicates the presence of a distributed object implementation where objects exchange messages and signals.The DC line indicates the possibility of generation from class diagrams. The DET line indicates the possibility of generation from a transition state diagram.

	Star UML	Papyrus	Yakindu	Modelio	VisualParadim	IBM RR
Version UML	2.0	2.5	-	2.4.1	2.0	2.4.1
DC	✓	✓	-	✓	✓	✓
DET	-	-	✓	✓	✓	✓
Operation	-	✓	-	RndTrip	RndTrip	✓
MOM	-	-	-	-	-	-
API Mapping	-	-	-	-	-	-
Round-trip	-	-	-	✓	✓	✓

Table 1: Code generation tools [1]

A look at the table 1 shows that, to date and to the best of our knowledge, no tool directly addresses distribution and communication issues or proposes an explicit mapping with the technical infrastructure. This observation leads us to question the way in which current tools meet the needs of developers in the field of distributed systems. In the absence of specific functionalities to manage these aspects, developers may encounter difficulties when attempting to model and generate code for distributed systems, which require communication between components to be taken into account.

In this paper we target the following problems:

- The distributed implementation of sending signals between objects,
- The abstraction of this implementation to obtain primitives.

In the rest of this article, we propose approaches aimed at integrating communications while remaining abstract about the protocol used (section 3), and we also carry out experiments in this area (section 6).

3 Contribution

Within the framework of predictive engineering, this article focuses on establishing a software production chain exploiting expressive models for distributed systems. The illustration is mainly done through automation systems, which encompass various concerns regarding distribution, communication, and control, and can be represented by various UML notation paradigms, while considering non-functional requirements. More specifically, we focus on programmable controllers that have an efficient execution environment, taking into account operational, security, and performance constraints. [9]. These constraints encompass general properties such as security and liveness, as well as characteristics specific to the environment or system, such as energy, danger, and quality of service. Furthermore, the scope can extend to software and hardware aspects. It is important to emphasize that the results presented are not limited exclusively to these systems.

To facilitate the code production chain, intermediate levels can be integrated by drawing inspiration from Model-Driven Development (MDD) approaches [6] and Software Product Lines (SPL) [5]. In the context of MDD development, it is essential to validate models before proceeding to the transformation and code generation steps. This approach helps reduce costs associated with late error detection [7]. Whether models are developed in UML, SysML, or any other architecture description language, they must be detailed enough to be executable. In addition to various verification techniques such as theorem proving or model checking, this paves the way for the possibility of testing these models.

In the context of manual development, experiments show that the generated code often does not strictly adhere to the models. The model is primarily used as a tool for interpreting and understanding the studied case rather than as a rigid guide for code semantics.

Our approach focuses on using model engineering to develop reliable PLC (Programmable Logic Controller) control software for EV3 robots. We adopt a model transformation method where the simplicity of the transformations is emphasized, while the complexity lies in the design of the transformation process, designed to be adaptable and configurable.
This approach will contribution on several levels :

- Integration of MQTT communication for robot coordination: Our approach integrates the MQTT protocol for communication between EV3 robots and other devices. Robots can then exchange messages efficiently and reliably, facilitating activity coordination and data exchange between robots and other connected devices.

- Automatic code generation: Once the models are specified, our approach automates code generation for EV3 robots. Using state machines, we translate model behaviour into executable source code. States represent the different situations the robot may encounter, while transitions describe the actions the robot can take to move from one State to another.

- Scalability of generated code: The generated code is designed to be modular and extensible to facilitate maintenance of applications for EV3 robots. Future researchers can add new features or modify robot behavior simply by modifying the models, without needing to revise all the source code. This enables quick adaptation to new needs.

4 Integration of communications

Before generating the models, it is necessary to establish the required programming infrastructure to facilitate communication between the robots. This initial configuration phase will offer users flexibility to allow them to opt for Wifi (LAN), MQTT protocol or Bluetooth as advanced means of communication. Otherwise, for users unfamiliar with this area, the system can autonomously determine the appropriate communication protocol based on the parameters available in the user's execution environment. Regardless of the user's level of expertise, the system can intelligently select the optimal communication mechanism, simplifying the configuration process. This integration of various communication protocols aligns with the principles of model-driven development (MDD), because it involves the systematic transformation of abstract communication models into concrete implementation details, ensuring that the chosen protocols are integrated into the overall system architecture.

4.1 Broadcast message send with MQTT

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe network protocol that enables devices to exchange messages across constrained networks, making it suitable for Internet of Things (IoT) devices where bandwidth and battery power are limited. MQTT functionality is similar to WebSockets but operates on top of the TCP/IP stack, and its design focuses on simplicity and a small code footprint [13]. The protocol allows clients to publish messages under a topic, and other clients can subscribe to the topic to receive these messages in real-time.

Eclipse Paho is the primary MQTT client library used in our project, enabling each robot to seamlessly connect to a cloud-hosted broker (MQTT Server). This architecture, centered around a cloud-based broker, facilitates multiple connections concurrently. To enable experimentation, we configured additional clients as applications (such as MQTT Explorer for desktop and MyMQTT for smartphones). These applications allowed us to interact with the robots by broadcasting messages to the broker in various formats, including raw text, XML, or JSON. The robots, operating with a custom leJOS program and already tailored to process incoming messages, consistently monitor for new messages. Upon receiving a message, the program on each robot interprets the command and initiates the corresponding operation, whether it involves navigation, object manipulation, or data collection tasks. The connection to the broker is secured with authentication to enforce a security layer, preventing unauthorized access.

Additionally, each robot can function as a client and broadcast messages, enabling peer-to-peer communication as well. For example, if Robot A wants to communicate with Robot B directly, it can publish a message to a specific topic that Robot B is subscribed to. This direct messaging capability facilitates real-time collaboration and coordination among robots without the need for intermediary communication through the broker.

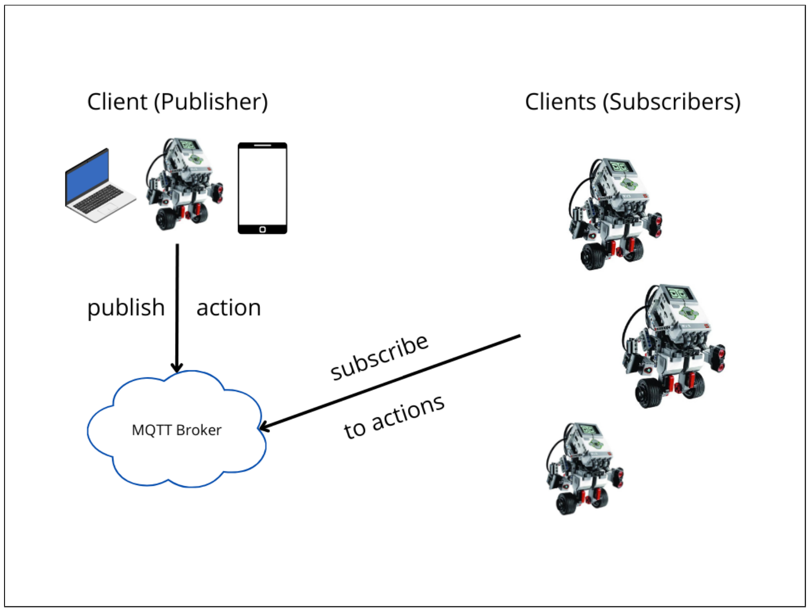


Figure 2: MQTT-Controlled Robotics Network

4.2 Peer-to-peer communication

Message send in object programming is between one sender and a receiver object. In an EV3 context, peer-to-peer message send is possible with Bluetooth or Wifi networks and protocols.

4.2.1 Bluetooth

Bluetooth in the context of the Internet of Things (IoT) is a wireless technology that allows the exchange of data between devices over short distances. It is commonly used for connecting and controlling smaller devices due to its low power consumption and relatively easy connectivity. In IoT ecosystems, Bluetooth can connect a wide range of devices, including smart home appliances like lighting systems and thermostats, wearable health monitors, security systems and more. Its coverage varies by device and version, with Bluetooth 5.0 offering improved range up to 240 meters under ideal conditions compared to earlier versions, enhancing its applicability in larger spaces and between multiple devices within IoT networks.

LeJOS offers comprehensive support that allows EV3 robots to communicate with each other or with other Bluetooth-enabled devices. It provides classes that facilitate Bluetooth connections, making it straightforward to pair and connect EV3 robots to other devices [3]. This includes scanning for available devices, establishing connections, and managing data transmission. Once connected, the framework allows for the sending and receiving of data packets between devices. This is typically done using streams (input and output), similar to standard Java socket programming. The EV3 can thus send commands or receive data from other devices to coordinate actions between multiple robots or interface with additional hardware.[4]

Bluetooth will be used as a second communication option for remote control of EV3 robots through LeJOS. We created a custom application on Android that connect to the EV3 over Bluetooth to send commands (simplified with a friendly user interface), which are then executed by the robot.

4.2.2 WiFi

In addition to the previous communications, EV3 robots also supports WiFi (LAN) communication through the LeJOS framework. This functionality enables communication between devices within the same local network, such as PCs or other robots. Remarkably, this communication can occur seamlessly, allowing for the execution of leJOS EV3 programs on a PC while accessing the hardware of a remote EV3 through a panel without any program modifications (LeJOS installed software is required for this to work). This operation ensures interoperability and flexibility in managing robotic systems especially in experimentation environment. An important aspect of this capability is the use of device discovery, which enables a leJOS EV3 program to locate EV3 units on the LAN automatically. Upon discovery, the program can establish communication with the identified EV3 unit, enabling streamlined collaboration and resource sharing across the network.

4.3 Abstracting and integrating message sending

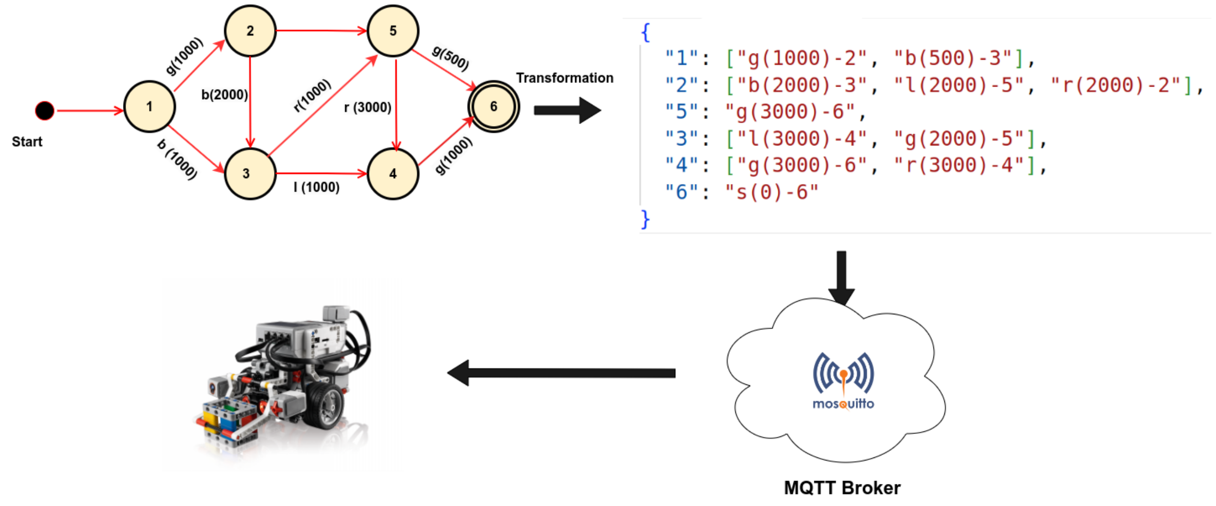
The EV3 robots are manipulated by programs, these programs are developed using the LeJoS framework[11], a Java development environment and the operating system installed in these robots (Can also be used in almost all LEGO Mindstorms robot models). LeJOS provides an API which allows access to the different components of a robot such as the LCD screen, motors, sensors... Programming is done from a development machine having Eclipse as a text editor, because a plugin of LeJOS is necessary to be able to compile and upload the program to the robot (via wifi or USB). In our case, we developed a program as part of experimenting with the operation of the MQTT protocol by transforming our robots into clients. In fact, the program, being uploaded to 2 or more robots, connects to a remote MQTT server and listens to messages that are published by one of our development machines (an MQTT client being installed). These messages must be in raw format and perform custom actions such as: "go" for forward, "stop" for stopping and "speedup" for increasing the speed of the motors. The obstacle detector was also used in this case. When it detects an object in front of it, the program sends a message to the MQTT broker to inform subscriber clients by stopping the engines.

This is to evaluate the response time of communication between the different robots connected to the broker as well as to integrate message-oriented middleware to reduce network dependence and simplify information sharing between clients.

5 Program Automation using State Machines

We started by formulating models using languages with heterogeneous semantics such as UML, using JSON parsers, and focus on programmable automatons, such as the Lego EV3, remotely controlled by programs written in Java deployed on Android. Although we do not achieve full automatic generation, our goal is to provide customizable assistance that helps the software designer streamline their development process.

In figure 3, the life-cycle of the program is presented in a system guided by the automaton. A visual modeling tool is used to design processes in the form of state diagrams (Action, Transition). From this modeling, a complete JSON representation of the activity process is obtained. It contains all the instructions required to control the program execution in the EV3. This description is then transmitted by the robot via the MQTT protocol. A linked list is then used to store the state and actions that need to be executed at a given time.



```
{
  "1": ["g(1000)-2", "b(500)-3"],
  "2": ["b(2000)-3", "l(2000)-5", "r(2000)-2"],
  "5": "g(3000)-6",
  "3": ["l(3000)-4", "g(2000)-5"],
  "4": ["g(3000)-6", "r(3000)-4"],
  "6": "s(0)-6"
}
```

Figure 3: "Automaton transmission process"

This approach enables a structured representation of the behavior of the EV3 robot. By defining a set of distinct states and transitions between these states based on specific events (actions), it is possible to accurately model the various sequences of actions possible for the robot. Once the automaton is modeled and transmitted to the robot, the corresponding code is automatically generated, eliminating the need to write complex code and significantly reducing syntax errors. This intuitive approach allows individuals with little or no programming experience to create complex programs for the EV3 robot using the automaton.

The automaton is defined based on 3 main parts:

- **Action :** Actions correspond to the logic of tasks executed at the level of the EV3 robot. They can appear within states and transitions. An action can be used to interact with the entire system, such as motors and various onboard sensors.
- **State :** States correspond to the steps in an activity process. A state identifies a specific point within the process. The actions that can be taken by various parties are fully defined by the set of outgoing transitions. A state can have an entry action and an exit action.
- **Transition:** represents a change in the state of the process. It connects two states, a source state from which it exits and a target state into which it enters. A transition corresponds to an action that is performed in response to an event. Additionally, transitions can have guards. These guards are checked, and the transition is executed only if they are true. Only one transition from a state is executed in response to an event.

6 Robotic System Development and Process Generation

Our model-driven approach is designed to facilitate the development process by abstracting intricate system components such as communication protocols (e.g., MQTT, Bluetooth) into reusable models. This method enables the seamless integration of diverse subsystems like motor control and sensor management for robotic vehicles while maintaining the logical structure of the original models. It streamlines the development lifecycle, promoting coherence between design models and their implementation.

6.1 Communication synchronization:

An advanced solution enabling usage of both MQTT and Bluetooth communications in parallel to control a robot has been implemented. This integration was achieved by the development of an Android application which interacts directly with the robot via Bluetooth, while offering the possibility to another client, based on the MQTT protocol, to control the same robot through network commands. Although two communication methods are synchronized, commands from MQTT are always given priority, with instructions via Bluetooth queued when necessary to avoid concurrency.

For example, this allows an engineer to design a system where control of a robot can be assigned to a specific Android client without explicitly predefining the communication technology to be used. Depending on operational conditions or specific requirements, the system can dynamically opt to use MQTT or Bluetooth. This will make it possible to switch between two communication modes as needed and considerably increase the robustness and responsiveness of the control system. To achieve this flexibility, we have implemented a software architecture where commands can be sent either by Bluetooth or by MQTT (but not both), with ease of switching which allows the communication mode to be adapted depending on the needs (like network conditions, device availability...). This model therefore offers a significant advantage in environments where conditions can vary rapidly, such as in industrial applications where reliability and precision are important.

The goal is to make the conception of the automatons simple, by creating them from designed models. The automatons, once configured, automatically generate Java code which maintains the same logical structure as the one initially developed. The models generated from the automaton

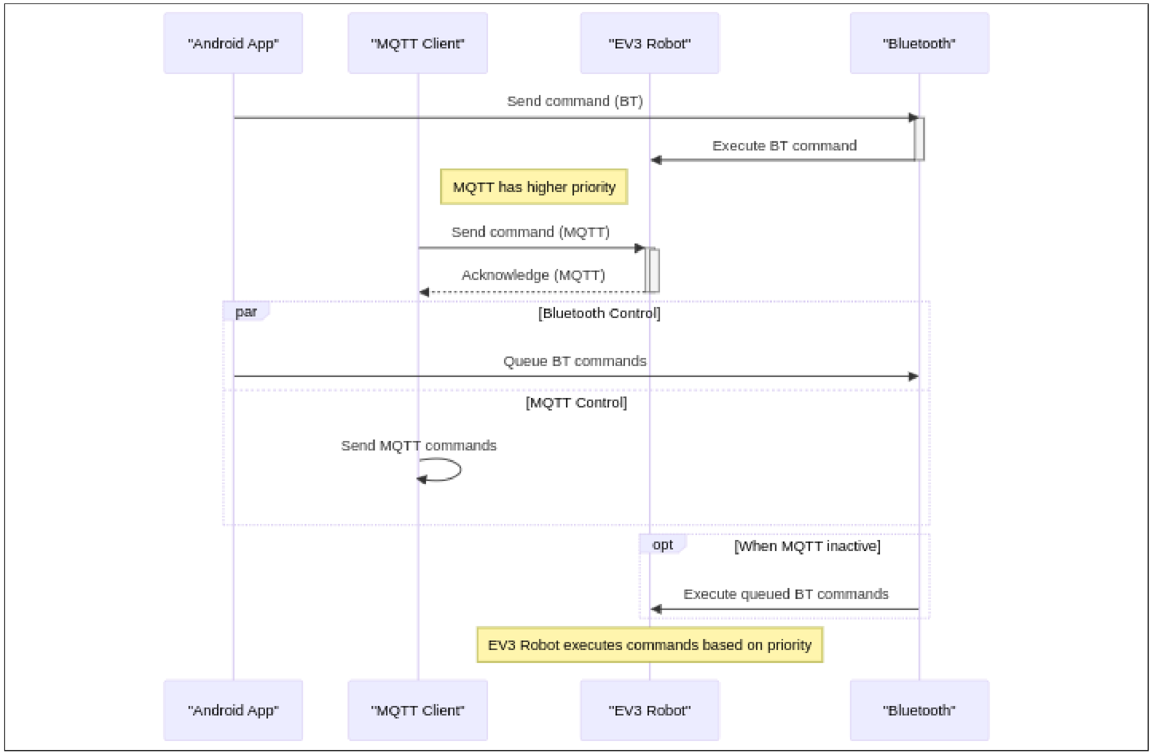


Figure 4: Android application functionality design

will include different types of diagrams such as class, sequence, state and flow diagrams, to make sure that the structure and behavior of applications during development was not modified.

In this context, forward engineering plays a crucial role by automatically generating code from the models specified in the diagrams. This process ensures that the system design remains aligned with its implementation. To guarantee consistency in program behavior between abstract models and manually written code during runtime, reverse engineering is used to extract code from existing models. This approach facilitates the seamless transition between design and implementation phases, maintaining coherence throughout the development lifecycle.

6.2 Automaton design

In parallel with the team responsible for synchronized communication, we had to design the automaton which will allow different activities to be executed with a single program as mentioned previously. Since the automaton will be uploaded in json format to the robot, it must respect a certain syntax.

The json file must have only one object containing the states, each state having an array of transitions:

```
[
    {
        "STATE_NUM" : [T1, T2, T3 ...],
        Other states...
    }
]
```

- `STATE_NUM`: the declared state which is always an integer (minimum is 0).
- `T`: the transition from the actual state to another state. It must also follow a specific syntax: `"ACTION(DELAY)-TARGET_STATE"`
  - `ACTION`: the operation to perform, which could be **`"g"`** for moving forward, **`"b"`** for the opposite, **`"s"`** to stop, or **`"r"`** and **`"l"`** for rotating right or left respectively.
  - `DELAY`: the time of the transition in milliseconds to perform the corresponding action. Must be included within parentheses.
  - `TARGET_STATE`: Preceded by a dash **`"-"`**, the target state where the transition is pointing and ending.

The start entry is pointing to the first state object by default.

Listing 1: Automaton example in json format

```
[{
    "1": ["g(1000)-2", "r(500)-3"],
    "2": ["s(0)-2"],
    "3": ["g(1500)-4"],
    "4": ["s(0)-4"],
}]
```

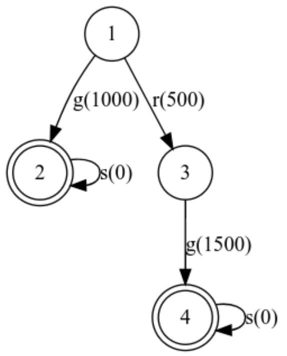


Figure 5: The corresponding automate diagram to Listing 1

From this simple automaton, there are two possible set of operations:

- (1 then 2): where the robot will move forward for 1 second and stops.
- (1, 3 then 4): where the robot turn right for 500ms, moves forward for 1 second and 30ms and finally stops.

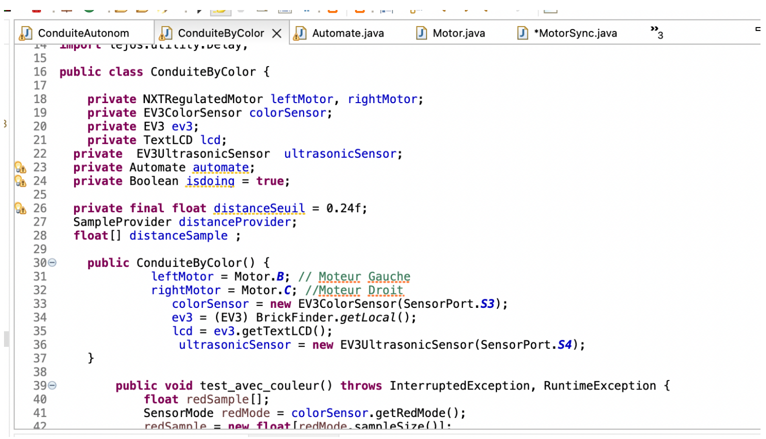
Indeed, the effectiveness of the automaton is not limited to such an example and can be much more complex than that by adding several states and transitions or by performing loops. All this without modifying a single line of the LeJOS program code since it remains intact. In our case, we used `HashMap<Key, Value>` in Java to model the state machine. Thus, each state represents a key, and the value can be composed of an array containing one or more values representing the transitions to execute and to switch to the next state..

We also had to check if the received file corresponds to the expected semantics, defined in the program, using the following regular expression pattern:

```
([a-zA-Z]+)\\((\\d+)\\)(?:-(\\d+))?
```

The automaton will be received by the LeJOS program in the EV3 robot and waits for a set of ordered states to execute.

In figure 6, Classes has been developed to manage the right and left motors, as well as for their synchronization. For example, if in the state machine we want to turn left, the program responsible for that action handles the task. Similarly, if we want to stop the robot, we synchronize the two motors to stop simultaneously, with another part of the program managing this function.



```java
import lejos.utility.Delay;

public class ConduiteByColor {

    private NXTRegulatedMotor leftMotor, rightMotor;
    private EV3ColorSensor colorSensor;
    private EV3 ev3;
    private TextLCD lcd;
  private  EV3UltrasonicSensor  ultrasonicSensor;
  private Automate automate;
  private Boolean isdoing = true;

  private final float distanceSeuil = 0.24f;
  SampleProvider distanceProvider;
  float[] distanceSample ;

    public ConduiteByColor() {
             leftMotor = Motor.B; // Moteur Gauche
             rightMotor = Motor.C; //Moteur Droit
                colorSensor = new EV3ColorSensor(SensorPort.S3);
                ev3 = (EV3) BrickFinder.getLocal();
                lcd = ev3.getTextLCD();
                 ultrasonicSensor = new EV3UltrasonicSensor(SensorPort.S4);
    }

        public void test_avec_couleur() throws InterruptedException, RuntimeException {
            float redSample[];
            SensorMode redMode = colorSensor.getRedMode();
            redSample = new float[redMode.sampleSize()];
```

Figure 6: Color tracking LeJOS program using the color sensor component

For sensors and robot components, we can directly incorporate them into the state machine, such as color sensors, distance sensors or any other types components.

7 Discussion

As a result for our experimentation, the development of robot software using LeJOS have yielded significant insights into the creation of a model-driven approach. By initially focusing on crafting an executable automaton for the robots, we established a solid foundation for subsequent model design phases. This approach not only streamlines the development process but also ensures that the resulting models accurately reflect the operational behavior of the robots. Additionally, the implementation of a basic communication layer enables the execution of simple programs and facilitating interaction with the robot's environment based on user-defined models. These advancements pave the way for a more efficient and user-friendly robotic ecosystem, where model-driven methodologies let users translate abstract concepts into robotic behaviors. Moreover, by integrating forward and reverse engineering practices, we can continuously refine and optimize the system, fostering ongoing adaptation to evolving user requirements and technological advancements.

8 Conclusion and Perspectives

In this study, we explored various aspects of modern software engineering, particularly focusing on the development of distributed systems and the automation of code generation. While significant progress has been made in this research, there is still much to be done to address current challenges and anticipate future needs. Looking ahead, one could consider integrating emerging technologies such as artificial intelligence and machine learning to enhance model-driven development (MDD) techniques, as well as improving collaboration and interoperability among existing tools and frameworks.

Finally, it's clear that our work is not yet finished. We are still in the development layer and code abstraction is not yet possible. More work on this is needed to ensure a model that will perform consistently and accurately compared to the initial code.

References

[1] Pascal Andre and Yannis Le Bars. Conception assistée de contrôleurs d'automates depuis des modèles UML. In *MSR 2019 - 12ème Colloque sur la Modélisation des Systèmes Réactifs, Nov 2019, Angers, France*, Angers, France, November 2019.

[2] Pascal André and Mohammed El Amin Tebib. Refining automation system control with MDE. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, pages 425–432. SCITEPRESS, 2020.

[3] LeJOS EV3 API. Class bluetooth. `https://lejos.sourceforge.io/ev3/docs/index.html?lejos/hardware/Bluetooth.html`. Accessed: 2024-04-27.

[4] LeJOS EV3 API. Communications. `https://lejos.sourceforge.io/nxt/nxj/tutorial/Communications/Communications.htm`. Accessed: 2024-04-27.

[5] C. Atkinson. *Component-based Product Line Engineering with UML.* Addison-Wesley object technology series. Addison-Wesley, 2002.

[6] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition.* Morgan & Claypool Publishers, 2nd edition, 2017.

[7] Martin Gogolla, Jarn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and Systems Modeling*, 4(4):386–398, 2005.

[8] Jussi Koskinen. Software Maintenance Costs. Technical report, School of Computing, University of Eastern Finland, Joensuu, Finland, April 2015.

[9] L. Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. Taylor & Francis, 2013.

[10] P. Roques and F. Vallée. *UML 2 en action: De l'analyse des besoins à la conception*. Architecte logiciel. Eyrolles, 2011. (in french).

[11] SourceForge. lejos. `https://sourceforge.net/projects/lejos/`.

[12] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. The MK/OMG Press. Elsevier Science, 2008.

[13] Wikipedia. Mqtt. `https://en.wikipedia.org/wiki/MQTT`, 2024. Accessed: 2024-04-13.