Scuola di Ingegneria Industriale e dell’Informazione

Master of Science in Management Engineering

Modular UML statechart modelling for simulating the behaviour of production systems with integrated dispatching policies

Advisor:

Prof. Marcello Urgo

Candidates:

Bagnati Edoardo 221851

Del Bello Alessandro 217336

Academic year 2023/2024

Abstract

This thesis offers a comprehensive study on production system modeling using UML statecharts, with a specific emphasis on integrating dispatching policies to optimize part flow through machines and buffers. The primary focus is the development of an advanced, UML statechart for a controller that governs a manufacturing system, integrating dispatching policies, such as FIFO, LIFO, and EDD, in conjunction with the CONWIP release policy.

The study begins with an evaluation of existing statechart models already developed in the *Virtual Learning Factory Toolkit,* developed by Urgo M and Terkaj W., which implement release policies without integrating dispatching mechanisms. The research introduces a new framework involving the creation of enhanced, modular UML statecharts for manufacturing systems composed by buffers and machines, managed by a controller. These statecharts are specifically designed to be adaptable across diverse scenarios and compatible with various policies. A primary objective was to create models that enable users to implement different release and dispatching policies seamlessly, without requiring modifications to the statecharts themselves.

The simulation phase of this research utilized the Python-based Sismic simulation library, which supports both modeling and execution of UML statecharts. To validate the model's accuracy, simulation results from Sismic were compared with those from Siemens Plant Simulation, a widely recognized industrial tool, which confirmed the robustness and reliability of the statechart design. Experiments were conducted across varied conditions, specifically analyzing the response of dispatching and release policies to fluctuations in part flow and machine availability. The findings affirmed that the developed statecharts adapted effectively to these dynamic changes.

In conclusion, this thesis introduces a novel production system modeling approach by incorporating dispatching policies into UML statecharts, enhancing modularity and flexibility for research and industrial use. It also lays the foundation for future research on real-world systems, offering advanced tools for real-time process optimization.

Key-words: UML Statechart, modularity, controller, dispatching policy, simulation, manufacturing system.

Abstract in italiano

Questa tesi offre uno studio completo sulla modellazione dei sistemi di produzione tramite UML statecharts, con un'enfasi particolare sull'integrazione delle politiche di dispatching per ottimizzare il flusso attraverso macchine e buffer. L'obiettivo principale è lo sviluppo di UML statechart per un controller che gestisce un sistema di produzione, integrando politiche di dispatching come FIFO, LIFO e EDD in combinazione con la politica di rilascio CONWIP.

Lo studio inizia con una valutazione dei modelli di statechart esistenti già sviluppati nel *Virtual Learning Factory Toolkit,* creato da Urgo M. e Terkaj W., nella quale sono implementate politiche di rilascio senza l'integrazione di meccanismi di dispatching. La ricerca introduce un nuovo framework che prevede la creazione di statechart UML modulari per sistemi di produzione composti da buffer e macchine, gestiti da un controller. Queste statechart sono progettate specificamente per essere adattabili a scenari diversi e compatibili con varie politiche. Un obiettivo primario è stato creare modelli che permettano agli utenti di implementare diverse politiche di rilascio e dispatching senza dover modificare le statechart stesse.

Nella fase di simulazione di questa ricerca è stata utilizzata la libreria di simulazione Sismic basata su Python, che supporta sia la modellazione che l'esecuzione di UML statechart. Per validare l'accuratezza del modello, i risultati della simulazione in Sismic sono stati confrontati con quelli ottenuti con Siemens Plant Simulation, uno strumento industriale ampiamente riconosciuto, confermando la robustezza e l'affidabilità del design degli statechart. Gli esperimenti sono stati condotti in condizioni variabili, analizzando specificamente la risposta delle politiche di dispatching e rilascio a fluttuazioni nel flusso delle parti e nella disponibilità delle macchine. I risultati hanno confermato che le statechart sviluppate si sono adattate efficacemente a questi cambiamenti dinamici.

In conclusione, questa tesi introduce un nuovo approccio alla modellazione dei sistemi di produzione, integrando politiche di dispatching in UML statechart, migliorando la modularità e la flessibilità per la ricerca e l'uso industriale. Essa getta inoltre le basi per future ricerche su sistemi reali, offrendo strumenti avanzati per l'ottimizzazione dei processi in tempo reale.

Parole chiave: UML Statechart, modularità, controllore, politiche di dispatching, simulazione, sistema di produzione.

Contents

1. Introduction *8*

2. Literature review *10*

2.1. PLC 10

2.1.1. Structure 12

2.1.2. PLC in simulation 14

2.2. UML 18

2.2.1. Introduction to Standard UML 18

2.2.2. Definition of UML Statechart 19

2.2.3. Structure and Syntax of UML Statecharts 20

2.2.4. Use Case: ATM example 25

2.2.5. UML State charts applications and benefits 29

2.3. Control Policies 32

2.3.1. Overview of Control Policies 33

2.3.2. Scheduling and Dispatching Policies 36

2.3.3. Use of UML Statecharts for defining control policies 41

2.3.4. Case Studies 42

3. Methodology *47*

3.1. Initial Context 48

3.1.1. Initial Statecharts 49

3.2. New statecharts development 53

3.2.1. New Controller Statechart 54

3.2.2. New Machine Statechart 55

3.2.3. Buffer Statechart 57

4. Implementation *59*

4.1. Introduction to Sismic library 59

4.1.1. Key Features of Sismic 59

4.2. Hypothesis of the simulation 64

4.3. Simulation code 66

4.4. Simulation output 75

5. Validation *77*

5.1. Siemens Plant Simulation system 78

5.1.1. System composition 78
5.1.2. System elements....79
5.1.3. System summary 86
5.1.4. Final Output....86

5.2. Validation code 90
5.2.1. KPI Evaluation Code....90
5.2.2. Validation events order....95
5.2.3. Validation buffer level 99

5.3. Experiments 101
5.3.1. Experiments setting 101

6. Conclusion 104

7. Bibliography....107

8. List of figures.... 111

9. List of tables 113

1. Introduction

In the context of the rapid transformation of the industrial landscape, manufacturing companies are facing mounting pressure to optimize their production processes in order to meet the growing demands for efficiency, flexibility and reliability. Modern manufacturing systems are required to manage a multitude of complexities, including fluctuating demand and unpredictable machine unavailability, while ensuring the seamless flow of materials through multiple processing stages. The implementation of effective control policies represents a crucial step in achieving these objectives, as they are essential for the management of the movement of parts through machines and buffers. The implementation of effective release policies, which regulate the introduction of work into the system, and dispatching policies, which prioritize tasks in real time, are essential for maintaining efficient operations.

In response to this requirement, the thesis is built upon an existing simulation system that employs UML statecharts to implement release control policies. The objective is to enhance the statecharts by integrating both release and dispatching control policies. The process entailed a comprehensive examination of the existing statecharts and their subsequent redefinition to facilitate the implementation of dispatching policies. Modularity was a pivotal concept throughout the development process, ensuring that the statecharts are adaptable and reusable in different contexts and scenarios. A key requirement was to design statecharts that allow users to apply various release and dispatching policies without needing to make any changes to the statechart models themselves. Subsequently, a simulation model was implemented for the purpose of validating the accuracy and efficacy of the enhanced statecharts.

Statecharts are an excellent tool for modeling production systems, as they clearly and systematically represent states, transitions, and events. In this thesis, statecharts are employed to model the behavior of three critical components within a production system: the controller, the machine, and the buffer. The controller statechart, which is the central focus of the work, is designed to be modular and responsible for managing part selection and dispatching, ensuring that parts are processed in line with defined control policies. The machine statechart manages the operational states of the machine, such as idle, working, and failure modes, ensuring that parts are processed correctly without errors. Meanwhile, the buffer statechart oversees the flow of materials, guaranteeing efficient storage.

These statecharts are specifically designed to create a simulation system where the controller plays a pivotal role. The controller is responsible for making real-time decisions based on current system

conditions, such as implementing release and dispatching policies. This real-time decision-making is influenced by the actual state of other components, ensuring that the production process adapts dynamically to changes in the system and maintains smooth operation.

In order to validate the effectiveness of the proposed statechart-based model, the Python-based Sismic simulation library was employed to simulate a variety of operational scenarios within a manufacturing system comprising a buffer, a machine, and a controller. The system was tested under a variety of conditions, including fluctuating part arrival rates, machine breakdowns, and different dispatching policies, in order to assess the efficacy of the statecharts in adapting to these dynamic changes. The dispatching policies that were applied are first in, first out (FIFO), last in, first out (LIFO) and earliest due date (EDD). Subsequently, the simulation results from the Python-based model were compared with those from Siemens Plant Simulation, a widely recognised tool in industrial applications, in order to ensure the accuracy, consistency and reliability of the statechart-based model.

In the chapters that follow, the thesis will provide a detailed overview of the methodology used to design and implement the statechart-based simulation model. This includes the development of new statecharts for the controller, machine, and buffer, as well as the integration of dispatching policies into these models. The implementation of the simulation is then discussed, followed by an in-depth analysis of the results from the validation process.

2. Literature review

2.1. PLC

Automation has emerged as a transformative force across various industries, significantly enhancing efficiency, safety, and productivity. Defined as the use of technology to perform tasks with minimal human intervention, automation aims to reduce manual effort and streamline operations. Central to this revolution are Programmable Logic Controllers (PLCs), which serve as primary control units in numerous automated systems.

- **Definition**

A Programmable Logic Controller (PLC) is a sophisticated industrial computing system meticulously designed to monitor input devices, implement real-time decision-making based on a customized program, and control output devices for effective process management. Widely adopted across production lines, machinery, and automated systems, PLCs represent a robust and efficient solution for industrial automation, streamlining operations and enhancing overall productivity within various sectors [1].

- **Evolution of Automation Technology**

Programmable Logic Controllers were initially developed to replace traditional relay-based systems. While relay systems were effective, they required considerable manual intervention and frequent maintenance, which limited their overall efficiency. PLCs introduced a new level of flexibility, accuracy, and reliability in various sectors, including manufacturing, automotive, and energy. They control machinery, processes, and complex operations, enabling safer, more adaptable, and efficient processes that meet the stringent safety and performance standards of modern industrial environments.

Historically, the evolution of automation technology has been remarkable. Early systems relied on electromagnetic switches for control, which lacked the adaptability necessary for rapid changes in industrial processes. The advent of Supervisory Control and Data Acquisition (SCADA[1]) and Distributed Control Systems (DCS[2]) marked a significant leap forward, introducing centralized

[1] SCADA is a system of software and hardware elements that allows organizations to control and monitor industrial processes by directly interfacing with plant-floor machinery and viewing real-time data [36].

[2] A Distributed Control System or DCS is a computerized system that automates industrial equipment used in continuous and batch processes, while reducing the risk to people and the environment [37].

control, real-time monitoring, and data acquisition. Among these advancements, the launch of the first PLC, the MODICON 084, was pivotal, offering a modular and programmable solution that greatly improved the flexibility and efficiency of automated processes. Since then, PLC technology has further advanced, incorporating high-speed processing capabilities and compatibility with various programming languages, such as Ladder Logic and Structured Text [2].

- **Application**

Programmable Logic Controllers (PLCs) have become integral to various industrial applications, primarily due to their standardized programming according to the IEC 61131-3[3] specification. This standard defines several programming languages, including Ladder Diagram, Structured Text, and Sequential Function Chart (SFC), which facilitate the creation and maintenance of programs within complex industrial environments [3].

Moreover, PLCs can be effectively represented and reproduced using UML state charts, which enable simulations of control logic and system behaviours. This representation aids engineers in visualizing and validating system functionality prior to deployment, ensuring that the design meets operational requirements. Additionally, they have gained popularity in manufacturing systems for implementing policies that ensure compliance with operational standards while optimizing productivity.

The advantages of PLCs over traditional relay-based systems are significant. First, PLCs offer enhanced flexibility; programs can be modified or expanded without requiring physical alterations to the hardware. This adaptability is crucial in dynamic production environments where requirements frequently change.

Additionally, PLCs can execute complex control logic and making real-time decisions with speed and precision. This capability is vital in high-demand automated systems, where timely and accurate responses are necessary to maintain operational efficiency. In particular, they are utilized production systems for implementing policies that ensure compliance with operational standards while optimizing productivity. Furthermore, the reduced maintenance requirements of PLCs contribute to lower downtime and operational costs, enhancing their appeal in industrial applications.

As a result of these benefits, PLCs are widely adopted across various sectors that require precise, reliable, and efficient control mechanisms. Common applications include manufacturing lines,

---

[3] IEC 61131-3 is the first vendor independent standardized programming language for industrial automation [38].

processing plants, and assembly operations, where PLCs play an important role in optimizing performance and ensuring operational integrity.

2.1.1. Structure

The structure of a PLC is inherently complex, comprising several key components that collectively support and enhance the control process [2]. The system is illustrated below:

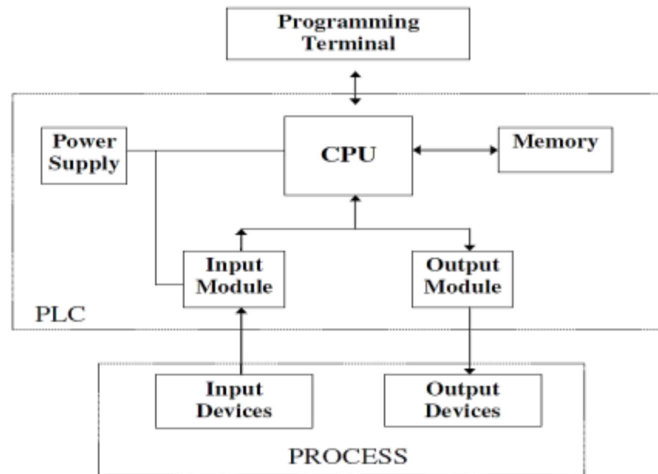

Figure 1: PLC structure

The core structure of a PLC comprises four essential components:

- The Central Processing Unit (CPU),
- Program Memory, Data Memory, and
- Input/Output (I/O) Modules.

The CPU is responsible for executing control instructions, making logical decisions, and ensuring smooth operations of the entire automated process. It continuously runs through a cyclic scan process, evaluating system inputs, running the control program, and updating outputs based on the program's instructions. This cyclical nature enables the PLC to monitor conditions and respond with high accuracy and consistency.

Program Memory is where the PLC stores the specific sequence of instructions (logic) that dictate the control flow. These instructions form the operational blueprint, guiding the PLC through various tasks and conditions specific to the automated process. Typically written in languages standardized by IEC 61131-3, such as Ladder Logic or Sequential Function Chart, the program memory enables

complex control sequences to be encoded efficiently and updated as required by the operational demands.

Data Memory is dedicated to storing status information for all input and output states, as well as intermediate values used by the control program. This memory area is crucial for real-time data storage, allowing the PLC to track changing system conditions and respond dynamically to the environment. For instance, values such as sensor readings, switch states, and interlock settings are recorded here, facilitating decision-making based on live data.

The Input/Output Modules connect the PLC to the physical world, receiving signals from input devices (like sensors, switches, and detectors) and sending control signals to output devices (such as motors, actuators, and alarms). Inputs provide information about the current state of the system, while outputs activate machinery, adjust operations, or alert operators. For example, if an emergency stop input is triggered, the PLC instantly halts operations, ensuring the safety of both the machinery and personnel.

To operate effectively, a PLC continuously performs a three-stage scan cycle: Input Scan, Program Execution, and Output Scan. During the Input Scan, the PLC reads and updates the status of each input device, storing this information in the input image table. In the Program Execution phase, the CPU processes each program instruction, updating an internal output table with values that correspond to the program's logical conditions. Finally, in the Output Scan, the updated output data is sent to the output modules, which then control the connected devices. This scan cycle, completed within milliseconds, enables PLCs to control processes in real-time.

PLCs are designed with robustness in mind, capable of withstanding challenging industrial environments that may involve high temperatures, vibrations, or electrical noise. Their modularity allows for straightforward configuration and expansion, making it easier to adapt to specific requirements, such as varying input/output needs, digital or analog signals, memory capacity, and processing speed. Selecting the appropriate PLC model thus involves considering parameters like the number of I/O points, type of I/O (digital vs. analog), CPU power, memory capacity, and compatibility with other system components. Additionally, manufacturer support, service availability, and cost-effectiveness are also significant factors in the decision-making process.

In summary, PLCs operate by continuously scanning inputs, processing user-defined logic, and updating outputs to control industrial equipment. This structure enables them to perform consistently in high-speed, repetitive operations, such as in manufacturing lines and chemical processing plants.

With advancements in standardization, especially IEC 61131-3, PLCs can now support multi-manufacturer compatibility, enhancing flexibility in complex automation systems.

2.1.2. PLC in simulation

The design of Programmable Logic Controllers (PLCs) often involves complex, discrete control systems that require a structured approach to development and analysis. UML (Unified Modelling Language) technology is instrumental in supporting this process, providing a set of standardized tools that help engineers analyse, model, and verify system requirements before implementation.

2.1.2.1. Introduction to Discrete System Development for PLCs

A discrete control system, such as a PLC, consists of a finite number of states and involves interactions between a control unit and a controlled object.

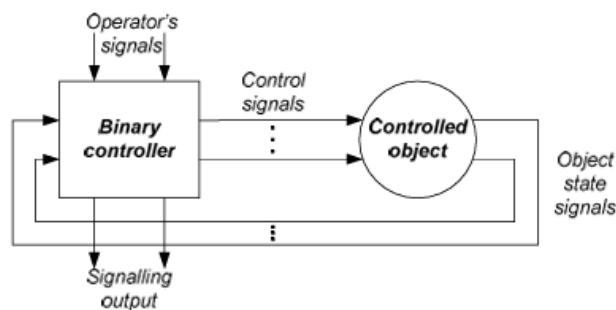

Figure 2: PLC logic

As showed in the figure, the control unit generates signals that command the controlled object, which in turn responds to these commands while feeding back information to the control system. Discrete control systems typically rely on binary signals, making them suitable for digital circuit implementations [4].

The development of such systems follows a systematic process represented in the figure below.

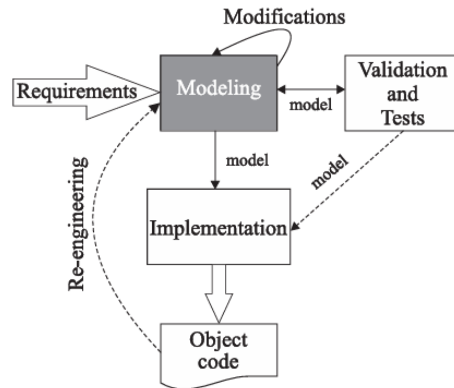

Figure 3: Construction process

Each stage plays a crucial role, with UML serving as a key tool, especially during the analysis and design phases. The outcome of these early stages lays the groundwork for the system's formal specifications, which guide later phases of formal verification and implementation [5].

Discrete System Development Process [5]:

1. *Analysis:* Focuses on requirements definition and feasibility, covering functional, technological, and economic factors. UML diagrams capture system requirements and interactions at this stage.
2. *Modelling:* Transforms initial requirements into formal specifications, often through finite state machines, Petri nets, and state chart diagrams, with symbolic verification of system behaviour.
3. *Implementation:* In this phase, the verified design is implemented using suitable technological resources and programs.
4. *Validation and tests:* Rigorous testing is conducted to verify functionality, performance, and compliance with standards.
5. *Maintenance:* The finalized system operates as required and can be modified based on evolving needs [4].

2.1.2.2. UML Modelling for PLC-Based Discrete Systems

In the context of PLC-based systems, UML modelling is primarily applied during the analysis phase. UML's standardized diagrams allow designers to create a simplified blueprint of the system under development, which can be further refined into formal models. Key UML techniques used in PLC

design include *object-oriented analysis, functional analysis,* and *behavioral analysis [4]*. Here are listed:

- **Object-Oriented Analysis**: In this stage, designers identify the real-world entities involved in the system, such as the controller, operator, sensors, and actuators. UML class and object diagrams are used to model these components, representing their attributes and relationships.
- **Functional Analysis**: This phase defines what the system must do, detailing each function the PLC performs. UML use case diagrams illustrate the relationships between system functionalities and help clarify the interactions between the PLC and the controlled object.
- **Behavioral Analysis**: Describes how each function is executed within the system, detailing the interactions and state transitions. UML activity, sequence, and state machine diagrams are commonly used, providing insight into the order and conditions under which processes occur.

2.1.2.3. Formal Models in PLC Design: Statechart Diagrams [4]

The transition from UML-based informal modelling to final system descriptions significantly enhances the representation of discrete control systems. Among the various formal modeling techniques, statechart diagrams stand out due to their effectiveness in capturing the dynamic behavior of state-oriented systems. Derived from UML state machines, statecharts illustrate how each condition or signal change triggers a transition from one state to another, enabling a detailed understanding of system functionality.

Statechart diagrams facilitate comprehensive verification through model checking, ensuring that the system operates reliably across all potential state configurations. This capability is critical in industrial applications where consistent performance is paramount. By incorporating Boolean expressions, statecharts are well-suited for direct implementation in digital controllers, allowing for a seamless transition from theoretical models to practical applications in the field.

Furthermore, the formalization of statechart diagrams empowers engineers to mathematically verify system behavior. This mathematical verification significantly increases the reliability of the control system by confirming that each function operates correctly in every state.

Once the system model is formalized, rigorous verification processes are employed. A primary method for state chart diagrams is model checking, which verifies that specific properties hold true across all possible states of the system. Unlike traditional testing methods that examine only a subset

of states, model checking guarantees comprehensive coverage, effectively identifying potential issues in different scenario such as deadlocks or livelocks that could disrupt system performance.

Collectively, these verification techniques provide engineers with the confidence that the PLC system is both logically sound and functionally reliable prior to hardware implementation. Following the verification phase, the PLC model is integrated into appropriate hardware or programmable logic devices, ensuring that the system can operate effectively in its intended environment.

This final implementation stage requires additional testing to confirm that the system behaves as intended under real-world conditions. In the maintenance phase, the PLC system operates according to its specifications, with provisions for updates and modifications in response to evolving operational requirements. This adaptability ensures the alignment and relevance of the system within dynamic industrial environments.

2.2. UML

2.2.1. Introduction to Standard UML

The Unified Modeling Language (UML), developed and maintained by the Object Management Group (OMG), is a standardized modeling language used in software engineering to visually represent a system's design. It offers a set of graphical notations for specifying, visualizing, constructing, and documenting the artifacts of software-intensive systems. UML provides organizations with the most effective means to capture, communicate, and leverage knowledge, ultimately helping them gain a competitive advantage through object-oriented design notation [6].

In particular, UML is a tool that assists software developers and system architects in communicating and understanding a system's design through various types of diagrams, including class diagrams, use case diagrams, and state charts. UML allows stakeholders to capture both the structural and behavioral aspects of a system in a standardized and consistent manner, facilitating its use across various research sectors [7].

Furthermore, It is a modelling language designed to articulate the characteristics of both new and existing systems, functioning independently of the project's domain, development process, and programming language, while being compatible with most object-oriented languages. While it contributes to development methodologies, it is not a methodology in itself. The language comprises a set of elements with graphical representations, embodying a semi-formal structure that combines natural language and diagrams to minimize ambiguity. It is governed by syntactic rules for creating valid models and semantic rules to ensure meaningful representation. Its method-independent nature allows for versatility across various development approaches [8].

Overall, UML is a powerful tool for modelling software systems. It enables stakeholders to collaborate effectively, analyze system requirements, and design solutions that meet the desired specifications. UML provides a wide range of diagrams, including [9]:

- *Class Diagrams:* illustrate the static structure of the system by representing classes, their attributes, methods, and relationships.
- *Use Case Diagrams:* depict the interactions between users and the system, capturing the functional requirements from the end-user perspective.
- *Sequence Diagrams:* Sequence diagrams show how objects interact in a specific order to accomplish a task, highlighting the flow of messages over time.

- *Activity Diagrams:* represent the workflow of a process, illustrating the sequence of activities, decision points, and parallel processes.
- *Statechart Diagrams*: model the dynamic behavior of an object by displaying its states and transitions in response to events.

2.2.2. Definition of UML Statechart

In the realm of UML, one of the primary diagram types is the *UML Statecharts*, also known as *UML state machine diagrams*. These diagrams serve as a form of dynamic modelling, allowing for the representation of a system's behavior over time by defining the flow and actions of systems as they evolve. They provide a graphical depiction of transitions between the states of an object or system, which may consist of one or more objects that communicate through events in response to both external and internal stimuli. Developed as part of the Unified Modeling Language (UML), statecharts offer a standardized approach to describe the dynamics and interactions within a system, particularly in the context of Discrete Event Systems (DES [4]) [9].

UML State charts possess a unique set of characteristics that make them an excellent tool for formalization. Below is a list of the key features [9] [10]:

- *Behavioral modelling*: UML Statecharts allow for a clear representation of how a system behaves in different states and how it reacts to events and conditions. These diagrams highlight the lifecycle of an object, showing how it transitions from one state to another based on specific conditions [11].
- *Constituent Elements:* A state diagram consists of states, transitions, events, and actions. States represent the conditions or situations in which an object can be, while transitions indicate the passage from one state to another in response to an event. Actions can be executed during these transitions or as part of the behavior of a state.
- *Support for Hierarchy:* UML Statecharts support the creation of composite states and orthogonal regions, allowing for the modelling of complex behaviors where a system can be in multiple active states simultaneously. This ability to represent the hierarchy of states offers greater flexibility in describing behaviors.
- *Verification and Validation:* One of the main advantages of UML Statecharts is their capacity to support verification and validation techniques. Using model checking tools, it is possible

[4] Discrete event simulation (DES) is a technique used to represent and study the behavior of physical systems, such as computer networks, that involve entities interacting over time [39].

to analyze the system's behavior to ensure that it meets specified requirements and that there are no errors in transitions or states [11].

- *Practical Applications:* UML Statecharts are widely used in various fields, including software engineering, industrial automation, and embedded system design. They are particularly useful for modelling interactive and reactive systems, such as smartphone applications, industrial control systems, and state machines.

In summary, UML Statecharts represent a powerful and versatile tool for modelling and designing complex systems. They provide a clear visual representation of system behavior and its interactions, facilitating communication among stakeholders and improving the understanding of system dynamics.

2.2.3. Structure and Syntax of UML Statecharts

As explained in last paragraphs, UML statecharts are graphical representations that illustrate the flow and behavior of systems. They consist of various symbols that convey the states of an object, transitions between these states, and the events that trigger these transitions. The standard documentation provided by the Object Management Group (OMG) details and updates all the standards utilized in UML diagrams, ensuring consistency and clarity in the representation of dynamic system behaviors. This comprehensive documentation serves as a critical resource for users and helps maintain uniformity in modeling practices across different projects and domains.

2.2.3.1. UML state chart elements

Each diagram consists of a series of elements that outline the flow of the system in response to both internal and external events. The main elements are as follows [9] [12]:

1) ***Events***: are the primary elements that trigger transitions and drive the flow of the system when an event occurs at a specific moment. Events represent changes or occurrences that the system responds to, such as receiving a message, a user action, or the passing of time. For example, in a manufacturing network, an event might be the “arrival” of a new order, which triggers the system to transition from an "Idle" state to a "Working" state.
2) ***States***: represent the various conditions or situations in which an object can exist at a given time, organized in different hierarchies.

They can be classified as follows:

- ***Single States:*** These are the basic states that indicate a specific condition of an object. For example, on a manufacturing line, a machine could be in a "*Working*" or "*Idle*" state, indicating that it is operational but waiting for the next task.
- ***Nested States:*** These are states that contain other states within them, allowing for a hierarchical representation of complex behaviors. For example, a "*Processing*" state may include nested states like "*Validating*" and "*Completing*." For instance, in a manufacturing line, a "*Production*" state may have nested states such as "*Assembly*," "*Quality Check*" and "*Packaging*", representing different stages of the production process.

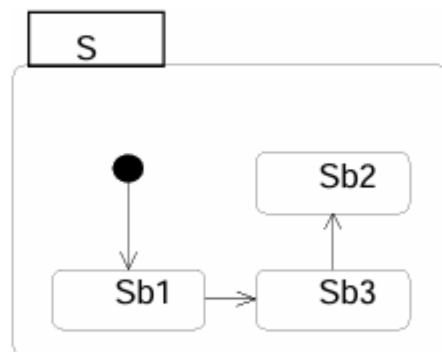

Figure 4: UML statechart nested states [9]

- ***Orthogonal States:*** These states enable an object to be in multiple states simultaneously, reflecting concurrent activities.

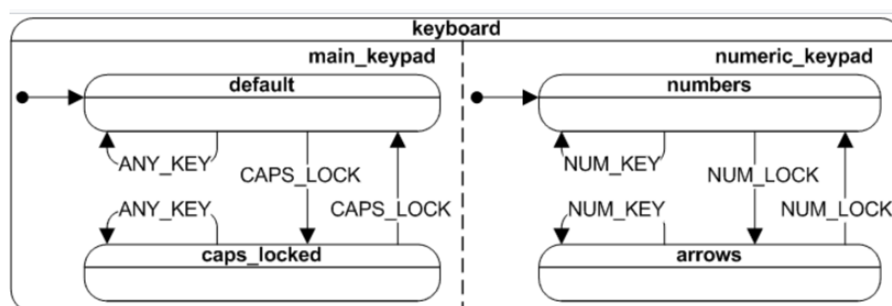

Figure 5: UML statechart orthogonal state [12]

As shown in the figure, the main and numeric keypads of a computer keyboard operate independently in different states. Orthogonal regions help avoid combining these states unnecessarily, keeping their behaviors separate.

3) ***Transition**:* are arrows that connect states, illustrating how the system moves from one state to another in response to events. They have a specific standard structure: above the arrow, the event trigger that leads from one state to another is indicated, followed by a slash and an action, which describes the activity occurring during the transition. Finally, guard conditions (Boolean expression) are placed in square brackets, which must be evaluated as true before the transition is activated. This notation provides clarity and precision in representing how and under what circumstances a system transition between states.

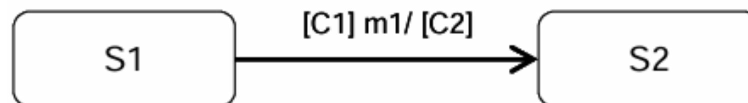

Figure 6: UML statechart transition structure [9]

When a transition occurs in UML, it follows a specific sequence of actions:

a. *Evaluate the Guard Condition:* The first step involves assessing the guard condition associated with the transition. The subsequent actions are executed only if the guard evaluates to TRUE.
b. *Exit the Source State Configuration:* If the guard condition is satisfied, the state machine exits the current state configuration, which can encompass multiple nested states.
c. *Execute Transition Actions:* After exiting the source state, the actions linked to the transition are executed.
d. *Enter the Target State Configuration:* Finally, the state machine enters the target state configuration, which may also consist of a hierarchy of states.

4) ***Actions**:* Activities that take place because of a transition or while in a specific state. They can be present as entry and exit action in a state or connected to a transition.

Additionally, the documentation of UML statecharts includes a series of auxiliary components that assist the user in representing the system in greater detail, such as:

- ***Initial State****:* Represents the starting point of the state machine, typically depicted as a filled black circle.

Figure 7: UML statechart initial state[9]

- ***Final State****:* Indicates the end of the state machine's execution, shown as a circle with a concentric filled circle inside.

Figure 8: UML statechart initial state [9]

There are situations where a simple direct transition between two states is inadequate to capture the necessary semantics. To address these cases, UML provides various pseudostates:

- ***Choice Pseudostate****:* Represents a decision point where transitions can lead to different states based on guard conditions, that could be numerical or string expressions.

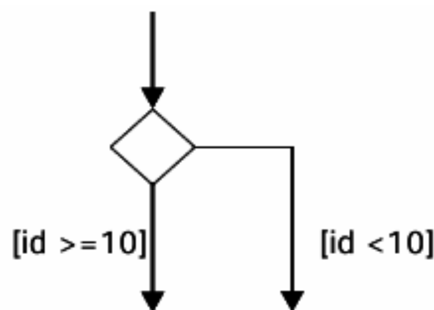

Figure 9: UML statechart choice pseudostate [9]

- ***Junction Pseudostate**:* Used to merge multiple transitions into a single one or split a transition into multiple paths.

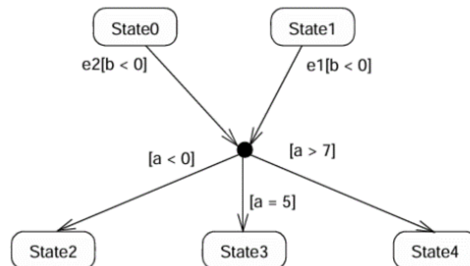

Figure 10: UML statechart junction pseudostate [9]

These components are not the only ones that enhance the functionality of state management in systems. There is also the *History State*, which is designed to remember the last active substate, allowing the system to return to a specific point in its state history. In addition, the *Deep History* feature tracks the entire path of nested states, providing a comprehensive understanding of the transitions that have occurred over time. The *Fork* component is crucial for splitting a transition into parallel paths, enabling multiple processes to occur simultaneously and improving overall efficiency. Conversely, the *Join* component serves to merge parallel transitions back into a single exit, ensuring a streamlined flow of control. Together, these elements provide greater precision and flexibility in system representation, allowing for more complex and dynamic behaviors to be modeled effectively.

2.2.3.2. Statechart sequence rules [12]

In Hierarchical State Machines (HSMs), the semantics of state transitions are inherently more complex due to the use of state nesting, entry and exit actions, and the management of active states. In UML statecharts, it is essential to understand that multiple states can be active simultaneously. For example, when a state machine is in a leaf state, all composite states that contain it, both directly and indirectly, are also considered active. This leads to a structured representation known as a state tree or state configuration, which captures the hierarchical nature of states.

When a state transition occurs, it follows a specific sequence of actions designed to ensure clarity and consistency in the behavior of the system. The process begins with the evaluation of a guard condition; actions associated with the transition are executed only if this guard evaluates to TRUE.

After this, the system exits the current state configuration, executes any transition-related actions, and finally enters the target state configuration.

Furthermore, UML differentiates between internal events and external events in state transitions. Internal events occur within the system and trigger state changes independently, such as a timer expiring. In contrast, external events originate from outside the system and cause transitions based on inputs, like a user pressing a button. This distinction is important for managing transitions: external transitions require leaving the current state, while local transitions can occur within the same state if the target is a substate.

A crucial concept in HSMs is the Run to Completion (RTC) model, which dictates that a state machine must fully process each event before starting on the next. During this processing phase, new incoming events are stored in an event queue, preventing interruptions. This ensures that the state machine

remains stable and avoids concurrency issues during event handling. However, this model also means that the responsiveness of the state machine can be affected by the longest processing step, as achieving shorter steps may introduce additional complexity into the design.

In summary, understanding the rules of sequence and execution within HSMs is fundamental for effective system modelling. By mastering the concepts of state configuration, transition types, and the RTC model, designers can create robust state machines capable of handling complex behaviours in a structured and efficient manner. This knowledge empowers teams to build systems that are not only functional but also responsive to real-time demands.

2.2.4. Use Case: ATM example

The ATM system case study [13], presented by Alexander Knapp and Stephan Merz, serves as an ideal example to demonstrate the application of UML Statecharts, showcasing the functionalities and characteristics of the elements discussed in previous chapters. The system consists of two primary actors: the ATM and the Bank, each exhibiting distinct behaviors that are modeled through separate state machines.

2.2.4.1. Paper Summary

In this case study, it is illustrated the approach using a simple UML model, as shown in Figure 1, which describes the interaction between an ATM, a bank computer, and a single user. The simulation focuses on the validation of the user's card and PIN, abstracted from the actual data and computations.

Below, the statecharts for the ATM (Fig. 11a) and the bank (Fig. 11b) are presented, specifying the dynamic behavior of each system component.

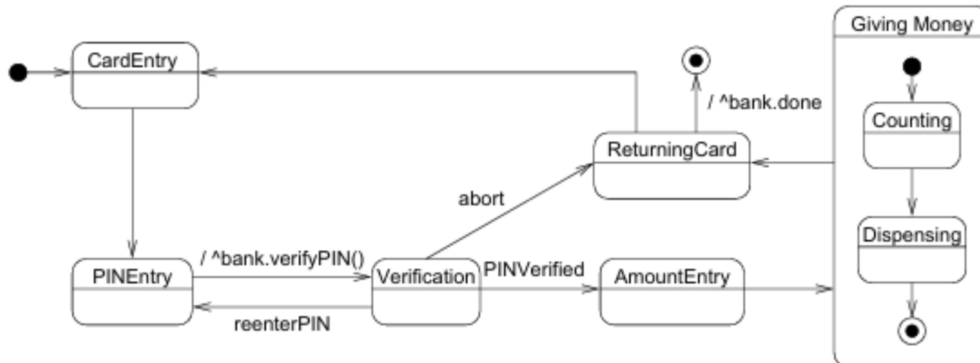

Fig. 11a: State machine diagram for class ATM

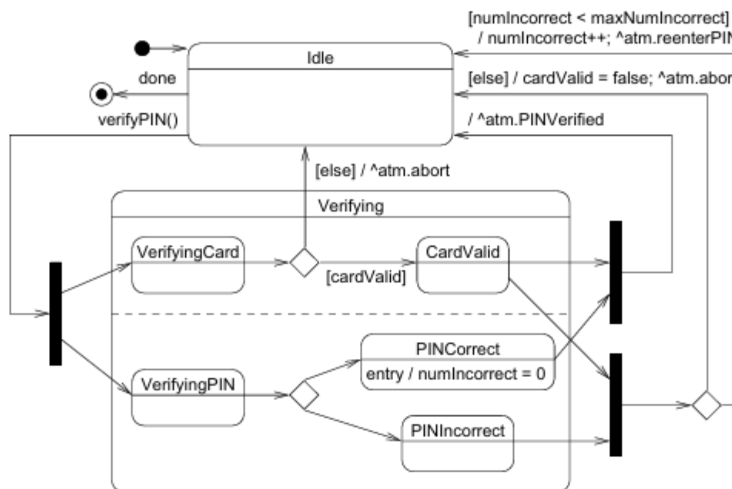

Fig. 11b: State machine diagram for class Bank

Figure 11: UML model of an ATM

System overview

In the context of UML statecharts, the representation of the various states of the system classes and their interactions through events provides a detailed overview of the operational dynamics. Let's take a closer look at the flow of information within the system, as depicted in the previous diagrams.

- **Process Initialization**

The flow begins when the user inserts the card and its corresponding PIN into the ATM. This action triggers an event signaling to the ATM to initiate the process of verifying the user's identity with the bank.

- **Identity Verification at the Bank**

Upon receiving the start event, the ATM transmits the information to the bank for user identity verification. This process occurs through an orthogonal state that handles both the validity of the card and the PIN.

- **Decision based on Verification**

Based on the verification result, the system makes a decision. If both the card and the PIN are valid, the system proceeds with the subsequent phases of the transaction. Otherwise, the system handles the error situation.

- **Error and Exception Handling**

In case the card or PIN verification fails, an "abort" signal is generated. This signal interrupts the transaction cycle and requires corrective action. For instance, if the entered PIN is incorrect, the ATM may send a signal to the bank to request a new PIN entry.

- **Corrective Actions and Retries**

In the event of an error, such as an incorrect PIN, an entry action is performed to handle the situation. The ATM may check the number of remaining incorrect attempts using an algorithm in the associated substate. If incorrect attempts are still possible, the user is allowed to retry. Otherwise, a new PIN entry is requested.

- **Transaction Continuation**

If all verifications are successfully passed, the transaction proceeds normally. The user can then select the desired withdrawal amount and complete the transaction successfully.

This flow of information, managed through the states and transitions of the UML statechart, along with algorithms processing user-entered information such as the card PIN and withdrawal amount, represents a crucial aspect of the ATM system's operation. Such information is processed and stored through algorithms implemented in the states, which use variables to perform checks and guide the flow within the system.

2.2.4.2. Statechart description

The dynamic behavior is specified by state machines for the ATM and Bank classes, as illustrated in Figures 11a and 11b. These statecharts outline the various transitions and states through which the system moves to handle the operations and interactions between the ATM, the bank, and the user.

State machines consist of different types of states, which are divided into simple, composite, and concurrent composite states. Simple states, like "*CardEntry*", represent single activities or operational phases. Composite states, like "*GivingMoney*", include substates that describe a sequence of related activities. Concurrent composite states, like "Verifying," contain orthogonal regions that represent the parallel composition of sub-machines, allowing multiple activities to execute simultaneously.

Each state has entry and exit actions (entry / numIncorrect = 0), which are executed respectively when a state is activated or deactivated. Additionally, states include activities (do-activities) that are performed as long as the state remains active, keeping the system engaged in specific operations.

Transitions between states are triggered by events, such as abort, and include guards, represented by conditions like [*cardValid*], which must be satisfied to allow the transition. During these transitions, actions to execute or events to emit, such as "*atm.abort"*, are specified, ensuring that the system reacts appropriately to received inputs.

Completion transitions, such as the transition leaving *CardEntry*, are triggered by an implicit completion event emitted when a state finishes all its internal activities. These transitions ensure that the logical flow of operations proceeds automatically once the actions in the current state are completed.

Fork and join states, represented by pseudo-states in the form of bars, synchronize transitions entering or exiting orthogonal regions, coordinating the start or end of parallel activities. Junction states, represented by diamonds, are used for branching cases, allowing the transition flow to diverge based on specific conditions.

This complex system of states and transitions allows for detailed modeling of the dynamic behavior of the ATM and the bank, ensuring accurate management of the various situations that may arise during daily operations.

This detailed case study on ATM management is an excellent example that clearly describes the execution flow and the characteristics of statechart elements, as well as how communication occurs between different statecharts through events. This detailed approach comprehensively illustrates how the ATM system interacts with the user and the bank during a transaction. It effectively handles both successful cases, allowing the customer to withdraw money securely and reliably, and error cases, intervening appropriately when issues such as an incorrect PIN or invalid card occur. Thanks to this effective and consistent management of information flows and operations, the ATM system ensures a smooth and reliable user experience in all situations.

2.2.5. UML State charts applications and benefits

2.2.5.1. **UML state charts application** [10][14]

The Unified Modelling Language (UML) has emerged as an important and versatile tool utilized across various engineering disciplines, thanks to its ability to effectively model complex systems with clarity and precision. In the context of manufacturing systems, discrete event simulations (DES) are crucial for optimizing production processes, enhancing operational efficiency, and reducing overall costs. Within this framework, UML state charts assume a significant role, providing a clear and structured methodology for modelling the dynamic behaviour of systems and illustrating the intricate interactions among various entities within those systems.

UML serves as a powerful tool for illustrating the states and transitions within a system, with each state reflecting a particular condition of its individual components as well as the overall system status at a given moment. The transitions delineate the shift from one state to another, instigated by specific events or conditions that arise during the system's operation. This modelling methodology is particularly advantageous for discrete event simulations in manufacturing, where a comprehensive understanding of the operational sequence and the interaction among various components is crucial for effective decision-making and optimizing processes. Additionally, UML is instrumental in

implementing dispatching policies that dictate the sequence of events within the system and facilitate real-time decision-making. This capability underscores its relevance in representing programmable logic controllers (PLCs), as discussed in previous sections. By harnessing UML state charts, engineers can effectively visualize intricate processes, foresee potential challenges, and devise solutions that contribute to enhanced productivity and operational efficiency.

2.2.5.2. Benefits of UML State charts

UML State charts are increasingly utilized due to the numerous benefits they offer, including enhanced clarity in modelling complex systems and improved communication among stakeholders. Here are the key advantages [14]:

a. *Detailed Representation:* UML state charts enable detailed representation of system states and transitions, offering a clear view of the system's dynamic behaviour. Each state and transition are explicitly defined, making it easier to understand and analyse the system's operations.
b. *Early Issue Detection:* The clarity provided by UML statecharts helps in early detection of potential issues. By modelling the system's behaviour accurately, engineers can identify inconsistencies, bottlenecks, or errors in the system's workflow before implementation. This proactive approach reduces the risk of costly modifications later.
c. *Improvement Identification:* Precision in modelling allows for a thorough examination of the system's processes. Engineers can pinpoint areas where efficiency can be enhanced, processes can be streamlined, or resources can be better allocated. This leads to more effective optimization strategies.
d. *Enhanced Communication:* UML state charts serve as a clear and common language among stakeholders, including engineers, managers, and operators. The precise nature of the diagrams ensures that all parties have a unified understanding of the system's behavior, facilitating better collaboration and decision-making.
e. *Modularity and Scalability:* The structured approach of UML state charts enhances modularity, making it easier to scale and adapt models as the system evolves. Whether adding new components or modifying existing processes, the precision in the state charts ensures that changes are seamlessly integrated without ambiguity. This modular design allows different statecharts to operate independently while still communicating effectively through events, thereby promoting a cohesive and efficient system architecture.
f. *Verification and Validation:* The precise nature of UML state charts aids in the verification and validation of the system model. Engineers can rigorously test the model against various

scenarios to ensure it behaves as expected. This verification process is crucial for maintaining the reliability and robustness of manufacturing systems. UML statecharts offer a high level of clarity and precision, which is crucial for accurately modelling the behaviour of manufacturing systems. This precision helps in identifying potential issues and areas for improvement in the system [15].

In conclusion, the effectiveness of UML statecharts in Discrete Event Simulation (DES) has been evaluated through various studies and practical applications. Key findings highlight their usability and benefits in improving the quality of simulations and the overall efficiency of manufacturing systems. Usability studies involving automation experts and engineers have shown that UML statecharts are easy to understand and use, providing a common language that bridges the gap between different engineering disciplines. From a cognitive science perspective, UML statecharts assist engineers in developing a clear mental model of the system, thereby enhancing their ability to predict system behavior and identify potential improvements. Moreover, the adoption of UML statecharts supports agile development practices by allowing for iterative refinement and continuous improvement of simulation models. This agility is crucial in responding to changing requirements and optimizing manufacturing processes in real-time. Overall, the integration of UML statecharts into DES practices fosters a more efficient and adaptable approach to system design and implementation.

2.3. Control Policies

Control policies in Production Planning and Control (PPC) are essential frameworks that define how resources are allocated, production tasks are organized, and scheduling activities are executed within manufacturing and service environments. These policies form the operational backbone of production systems, guiding them to efficiently balance demand fulfillment with optimal resource utilization. Given the growing complexity in global supply chains and market fluctuations, PPC policies play a critical role in navigating the intricate dynamics of production management, ensuring that organizations maintain both efficiency and adaptability in fulfilling customer demands. Effective control policies in PPC establish structured methodologies that drive decision-making processes across various time horizons, from long-term strategic planning to short-term operational execution [16] [17].

At the strategic level, PPC control policies serve as a blueprint for long-term planning, aligning production capacities, resource allocation, and workforce capabilities with overarching organizational goals. Strategic PPC policies encompass high-level decisions such as facility expansion, technology investments, and partnerships, all aimed at fostering a production system that can adapt to shifts in market demands and competitive landscapes. Meanwhile, tactical planning within PPC translates these strategies into actionable steps, managing resource distribution and adjusting production capacities to meet anticipated changes in demand. This mid-term planning horizon involves decisions on inventory management, workforce scheduling, and capacity adjustments that ensure responsiveness without incurring unnecessary costs [18] [19].

On the operational level, PPC control policies address immediate production tasks, scheduling, inventory control, and workflow management to maximize efficiency and meet real-time customer demands. Operational control is crucial for maintaining process continuity, achieving on-time deliveries, and managing the intricate interdependencies within production tasks. Decisions in this phase include job sequencing, machine assignments, and real-time inventory replenishment, which directly influence production efficiency and product quality [20].

The importance of PPC control policies has only grown with the advent of Industry 4.0 and the increased digitization of production processes. Technologies such as the Internet of Things (IoT), real-time data analytics, machine learning, and automation are transforming PPC by enabling data-driven decision-making and allowing manufacturers to respond more swiftly to dynamic market conditions [21]. For instance, real-time data can inform more accurate demand forecasting, minimize

waste, and enhance supply chain coordination, thus making PPC control policies a vital foundation for organizations aiming to remain agile and competitive in a rapidly evolving industrial landscape.

Studying control policies within PPC offers valuable insights into the methodologies and technologies that optimize production processes, ensuring that organizations can adapt to internal and external challenges effectively. In a globalized world where responsiveness and efficiency are paramount, PPC control policies provide the structured yet adaptable framework necessary for achieving operational excellence [22][23].

2.3.1. Overview of Control Policies

Control policies encompass a variety of methodologies that align production resources with demand, optimize scheduling, and ensure efficient workflow management. These policies include traditional systems like Material Requirements Planning (MRP), Workload Control (WLC), Kanban, Constant Work-In-Process (CONWIP), Theory of Constraints (TOC), and hybrid models like Paired Cell Overlapping Loops of Cards with Authorization (POLCA). More advanced, data-driven control policies, influenced by Industry 4.0 technologies, have also emerged, enhancing visibility, traceability, and adaptability within manufacturing environments [23][24].

There are various control policies, which can be broadly classified into *push* and *pull* approaches. In a *push* system, production is driven by forecasted demand, with parts or products pushed through the process according to a schedule. Conversely, in a *pull* system, production is initiated based on actual demand, with parts or products pulled through the system only when required. Below is a brief introduction to the most commonly used control policies within this framework.

- **Material Requirements Planning (MRP)**

Material Requirements Planning (MRP) is a push-based system that schedules production activities based on forecasted demand and manages material flows. MRP is widely implemented across various industries due to its ability to perform detailed material planning and align production schedules with demand forecasts. However, MRP is highly dependent on the accuracy of these forecasts and the stability of the production environment. In settings where demand is unpredictable or production variability is high, MRP can become inefficient, potentially leading to overproduction and excess inventory. MRP II, an advanced version, incorporates planning for labor and machinery to address some of these limitations [22].

- **Workload Control (WLC)**

Workload Control (WLC) operates as a pull-based system, regulating the flow of jobs onto the shop floor based on the current workload. WLC is particularly effective in environments characterized by high customization and variability, making it an excellent tool for reducing congestion and stabilizing lead times. By managing the workload at critical control points such as customer enquiry, job entry, and job release, WLC ensures that jobs are only introduced into production when the system has the capacity to handle them. However, the implementation of WLC can be resource-intensive, as it requires real-time data collection and sophisticated monitoring systems [16].

- **Kanban**

Kanban, another pull-based system, is widely used in lean manufacturing. It uses visual signals to trigger production activities based on real-time demand. Kanban excels in repetitive and standardized environments by reducing waste and aligning production closely with actual demand. However, in environments where product variability is high or workflows are complex, Kanban may lack the necessary flexibility to handle customization efficiently [24].

- **Constant Work-In-Process (CONWIP)**

Constant Work-In-Process (CONWIP) maintains a constant level of work-in-progress by releasing new jobs into the system only when completed jobs leave. This system is more flexible than Kanban, making it well-suited to environments with variability. CONWIP focuses on managing the total amount of WIP rather than specific part flows, which allows it to adapt more easily to changes in production. However, it requires careful management to avoid bottlenecks and idle resources, which can lead to inefficiencies [16] [24].

- **Scheduling and Dispatching Policies**

Scheduling policies are critical in production planning and control as they determine how tasks are prioritized and allocated to resources. Common policies include First-in First-out (FIFO), Last-in First-out (LIFO), Earliest Due Date (EDD), Shortest Processing Time (SPT), and Critical Ratio (CR). Each policy has specific strengths, such as minimizing lead times or ensuring on-time delivery. A dispatching rule is used within these scheduling policies to select the next job to be processed from a set of jobs awaiting service, allowing for real-time prioritization as resources become available. The

choice of scheduling policy depends on the production environment, resource availability, and operational goals, enabling an efficient organization of tasks to meet specific performance objectives [25].

- **Theory of Constraints (TOC)**

Theory of Constraints (TOC) focuses on identifying and optimizing bottlenecks, or constraints, within the production process. By improving performance at these critical points, TOC aims to increase the overall throughput of the system. It often utilizes the Drum-Buffer-Rope (DBR) mechanism to synchronize production with the system's constraints. While TOC effectively addresses bottlenecks to improve throughput, it may face challenges in environments where constraints frequently shift, requiring continuous adaptation [22] [24].

- **Paired Cell Overlapping Loops of Cards with Authorization (POLCA)**

Paired Cell Overlapping Loops of Cards with Authorization (POLCA) is a hybrid push-pull system designed for environments with high variability and customization. Combining the signaling benefits of Kanban with the flexibility of CONWIP, POLCA is particularly suitable for Quick Response Manufacturing (QRM) environments. It is highly effective in managing complex, non-repetitive workflows. However, POLCA requires sophisticated tracking and coordination across multiple production cells, making it more complex to implement than simpler systems [16] [22].

2.3.1.1. Advantages and Disadvantages of Control Policies

Control policies play a key role in streamlining production processes, enhancing efficiency and optimization by minimizing delays, reducing work-in-progress (WIP), and improving resource utilization. Additionally, policies like Workload Control (WLC), Constant Work-in-Progress (CONWIP), and Paired-cell Overlapping Loops of Cards with Authorization (POLCA) offer flexibility and adaptability, making them particularly suited for environments characterized by high variability or customization. These policies also contribute to improved delivery performance, stabilizing lead times and increasing on-time delivery rates. Furthermore, by effectively managing WIP and reducing waste, control policies contribute to lowering operational costs [24].

The implementation of many control policies, especially advanced systems like Workload Control (WLC) or Paired-Cell Overlapping Loops of Cards with Authorization (POLCA), can be complex and requires significant resources both for initial setup and ongoing maintenance. Moreover, these systems tend to be data-intensive, particularly in the context of Industry 4.0's smart production

planning and control (PPC), where sophisticated data management systems are essential, often leading to higher costs. Additionally, certain policies, such as Kanban, which rely on standardization, may be less effective in environments characterized by high customization or variability [23][24].

Control policies are essential for efficient and responsive manufacturing operations. The choice of the appropriate control policy depends on the production environment, the level of variability, and the specific operational goals. With the integration of Industry 4.0 technologies, these policies are becoming smarter, more dynamic, and capable of handling the complexities of modern production environments. However, each policy comes with its own set of challenges, requiring careful consideration of operational constraints and technological capabilities.

2.3.2. Scheduling and Dispatching Policies

Scheduling policies are fundamental to the effective management of production systems within the framework of Production Planning and Control (PPC). These policies define the rules and methods for allocating resources, prioritizing tasks, and determining the sequence in which production activities are executed. A critical subset of scheduling policies are dispatching policies, which specify the rules for selecting and sequencing jobs at the moment a resource becomes available. In dynamic production environments, where demand fluctuates and resources are limited, dispatching policies play a vital role in enabling real-time decision-making that directly influences production flow and efficiency [26].

The primary objective of scheduling and dispatching policies is to optimize key performance indicators (KPIs) such as throughput, resource utilization, work-in-progress (WIP), and lead times. Dispatching policies, in particular, focus on prioritizing jobs based on specific criteria, such as earliest due date, shortest processing time, or critical ratio, ensuring that production operations remain smooth, predictable, and aligned with customer timelines. Effective dispatching policies reduce bottlenecks, balance workloads across resources, and help minimize delays, ultimately supporting the efficient operation of complex production systems [25]. In high-variability settings, dispatching policies provide the flexibility needed to respond dynamically to real-time production conditions, often prioritizing tasks based on due dates, processing times, or job importance [26].

Scheduling and dispatching policies operate across different time horizons, tailored to the specific needs of production processes. Short-term scheduling focuses on the immediate sequencing of jobs, machine assignments, and prioritization, aiming to optimize daily or hourly operations. Dispatching policies are crucial here, providing real-time instructions on which task to process next as resources

become available. Policies like Shortest Processing Time (SPT), Earliest Due Date (EDD), and Least Slack Time (LST) ensure that tasks are sequenced effectively to meet immediate operational demands [26].

Medium-term scheduling policies are more tactical, balancing resources over weeks or months. In this context, dispatching policies help maintain a steady production flow by prioritizing tasks in real-time to prevent idle time or resource overloading. For example, Critical Ratio (CR) dispatching adjusts priorities dynamically based on each task's urgency relative to available resources. This level of scheduling also manages batch sizes, coordinates multiple job streams, and supports adjustments to workforce schedules, creating a stable, balanced production environment [19].

In long-term scheduling, policies align with strategic objectives, ensuring that the production system can meet future demand through workforce training, preventive maintenance, and resource allocation planning. Although dispatching policies operate predominantly in short-term and tactical contexts, their effectiveness contributes to the success of long-term scheduling by reducing cumulative delays and maintaining a stable production process [26].

The rise of Industry 4.0 has further advanced the role of dispatching policies, enabling data-driven and adaptive approaches to real-time task prioritization. By integrating technologies like IoT, real-time analytics, and AI-based algorithms, dispatching policies now allow for dynamic adjustments in job sequencing and resource allocation, based on live production data. This data-driven dispatching optimizes sequencing to prevent bottlenecks, reallocates resources, and rapidly adapts to changing production demands, improving overall system responsiveness and efficiency [27].

In summary, dispatching policies form the backbone of real-time decision-making in scheduling, ensuring that production remains efficient, responsive, and capable of adapting to complex, fluctuating environments. They work in tandem with scheduling policies across various time horizons, playing an essential role in balancing production efficiency with flexibility. The next section explores specific dispatching policies, examining how each approach contributes to achieving optimal performance under different production constraints.

2.3.2.1. Key Dispatching Policies

Dispatching policies define the rules for determining:

- Which task should be selected as the next job to be processed from a set of jobs awaiting service when a resource becomes available.

- How tasks are prioritized based on factors such as due dates, processing times, or resource availability.

Effective dispatching policies aim to improve key performance indicators such as throughput, utilization, on-time delivery, and work-in-progress (WIP). Different industries and production environments use specific scheduling policies based on their unique constraints and objectives.

- **First-in, First-out (FIFO)**

One of the most common dispatching policies is First-in, First-out (FIFO). This policy processes tasks in the order they arrive, without considering processing time or due dates. The simplicity of FIFO makes it easy to implement as it requires minimal decision-making. However, its lack of prioritization often leads to inefficiencies, especially when there is a mix of short and long tasks. In such cases, shorter jobs may be delayed by longer tasks, resulting in high lead times and suboptimal performance in dynamic environments [22] [24] [28].

- **Last-in, First-out (FIFO)**

The Last-in, First-out (LIFO) policy prioritizes the most recently added tasks, processing items in reverse order of arrival. This approach benefits scenarios where tasks added later are more urgent or perishable, ensuring they are addressed promptly. While straightforward to implement, LIFO can lead to starvation of earlier tasks if new items consistently enter the system, potentially resulting in neglected tasks or inefficiencies in environments with mixed task priorities and durations.

- **Earliest Due Date (EDD)**

The Earliest Due Date (EDD) policy prioritizes tasks based on their deadlines, making it particularly useful in environments where meeting delivery dates is critical. EDD minimizes tardiness by ensuring that tasks with the closest deadlines are processed first. However, this can result in longer processing times for tasks with later due dates, potentially extending the overall make span and delaying lower-priority tasks [24] [28].

- **Shortest Processing Time (SPT)**

Shortest Processing Time (SPT) gives priority to tasks with the shortest processing times. This policy aims to reduce the total flow time and minimize work-in-progress (WIP) by quickly completing

smaller jobs. While SPT effectively improves throughput, it can also lead to "starvation" of long jobs if shorter tasks keep arriving, and it may fail to prioritize urgent tasks [24] [28].

- **Least Slack Time (LST)**

Least Slack Time (LST) assigns priority based on the time left before a task's due date relative to its remaining processing time. Tasks with the least slack time, those at greatest risk of being late, are processed first. This policy balances the need to meet deadlines with the time required to complete tasks, reducing both tardiness and bottlenecks. However, LST is complex to implement, requiring continuous monitoring of due dates and processing times, which can lead to inefficiencies in environments where job priorities frequently change [23] [28].

- **Critical Ratio (CR)**

The Critical Ratio (CR) policy calculates a priority index for each task by dividing the time remaining until the due date by the processing time. If the CR value is less than one, the task is behind schedule; if it's greater than one, there is still buffer time. CR is dynamic and responsive, adjusting priorities based on task urgency, but it requires real-time data and frequent recalculations, which makes it challenging to manage in fast-paced production settings [24][28].

- **Weighted Shortest Processing Time (WSPT)**

In Weighted Shortest Processing Time (WSPT), tasks are prioritized based on their processing times and assigned weights, which usually reflect the task's importance or the cost of delay. WSPT provides flexibility in balancing both short and high-priority tasks. However, determining appropriate weights for each task can be difficult, and this policy may not always ensure that critical jobs are completed on time [22] .

- **Round Robin (RR)**

Round Robin (RR) assigns each task a time slice (or time quantum) during which it can be processed. When the time slice expires, the task moves to the back of the queue and the next task is processed. This continues until all tasks are completed. RR fairly allocates resources across tasks, preventing any single job from monopolizing the system. However, it can lead to inefficiencies if tasks have widely varying processing times, as each job receives the same amount of time regardless of its actual needs [23] [24] [28].

- **Johnson’s Rule**

Johnson’s Rule is used for scheduling jobs that must pass through two machines in the same sequence. It determines the optimal sequence to minimize total completion time, with jobs that have the shortest processing time on either machine being sequenced first. Johnson’s Rule minimizes make span for two-machine systems and provides an efficient schedule by reducing idle times between operations, but it is limited to two-machine scenarios and is unsuitable for more complex systems [24] [28].

2.3.2.2. Advanced Scheduling in Industry 4.0

With the rise of Industry 4.0 and smart manufacturing, scheduling policies have become more dynamic, and data driven. Real-time data collection, enabled by cyber-physical systems (CPS) and the Internet of Things (IoT), allows for more responsive and adaptive scheduling. Some of the advanced scheduling approaches emerging in Industry 4.0 include:

- **Smart Scheduling**: Algorithms that use real-time data from sensors and machines to adjust schedules dynamically based on current production conditions, machine availability, and job progress [23].
- **Artificial Intelligence (AI) and Machine Learning (ML)-based Scheduling:** These systems predict bottlenecks, optimize job sequencing, and allocate resources more efficiently by learning from historical data [23].
- **Autonomous Scheduling Systems:** In these systems, decision-making is decentralized, with machines and production cells able to schedule tasks autonomously based on local conditions and priorities [23].

2.3.2.3. Advantages and Disadvantages of dispatching policies

Well-designed dispatching policies enhance efficiency and throughput by optimizing resource utilization and reducing idle times, which in turn improves the overall system performance. Additionally, advanced scheduling systems offer customization and flexibility, allowing them to be tailored to meet specific operational objectives, such as minimizing lead times, reducing work-in-progress (WIP), or prioritizing urgent tasks. In the context of Industry 4.0, these scheduling policies also have the capability to adapt in real time based on dynamic data, ensuring that resources are consistently allocated in the most efficient manner.

However, implementing advanced dispatching policies, especially those that utilize real-time data and artificial intelligence, can be complex and requires sophisticated systems along with constant monitoring, making them expensive to set up and maintain. Moreover, these policies are highly dependent on accurate real-time data, meaning that robust data collection and management systems are essential. Any disruption in data flow can lead to inefficiencies or delays in production. Additionally, there are trade-offs to consider, as many scheduling policies tend to optimize one objective, such as reducing lead times, at the expense of others, like balancing workloads or meeting due dates. Achieving the right balance requires careful planning and consideration of these competing priorities [23] [24].

Scheduling and dispatching policies play a vital role in the smooth operation of manufacturing systems. From simple policies like First-Come, First-Served (FCFS) to advanced, dynamic approaches driven by Industry 4.0 technologies, scheduling strategies are integral to optimizing production, reducing bottlenecks, and improving key performance metrics. However, the complexity and resource requirements for implementing these systems must be carefully considered, especially in more advanced, data-driven contexts.

2.3.3. Use of UML Statecharts for defining control policies

In complex systems where dynamic task management and state transitions are essential, UML statecharts are highly effective for designing control policies. They provide a structured, visual tool for modeling the various states of a system and the events that trigger transitions between those states. By implementing UML statecharts, system designers can clearly define task lifecycles, improving how resources are allocated and how workflows are managed throughout production. This modeling approach makes it easier to visualize and control the flow of work, particularly in systems where tasks must transition smoothly between different operational states [29] [30].

UML statecharts are particularly valuable in real-time systems where tasks must meet strict timing requirements. In such systems, a run-to-completion scheduling model ensures that once a state transition is initiated, the system completes the current task before processing subsequent events. This prevents conflicts between overlapping operations and guarantees that tasks are executed predictably, an essential feature for maintaining efficiency in time-sensitive environments. UML statecharts simplify this process by clearly defining how and when transitions should occur, ensuring smooth system operation without the risk of unpredictable behaviors [30].

The hierarchical nature of UML statecharts further aids in managing complexity. Systems can be divided into states and substates, creating a detailed representation of a machine's operational phases. For example, a machine may switch between high-level states such as "idle" or "processing" with each of these states having additional substates. This layered approach provides flexibility in design, enabling engineers to oversee multiple processes concurrently, such as monitoring the system's health while processing tasks [29].

Moreover, UML statecharts allow for dynamic adaptation of scheduling policies in real-time. Through the use of guard conditions and state variables, the controller can modify its behavior based on the system's current workload or other operational factors. For instance, a dispatching controller could implement policies like Earliest Due Date (EDD) or Shortest Processing Time (SPT) to prioritize tasks as needed, adjusting in real-time as system demands change. This adaptability makes UML statecharts a practical tool for addressing varying production requirements without needing to redesign the system from the ground up [29] [30].

In addition, UML statecharts are highly modular, allowing components to be reused and scaled across different systems. This modularity ensures that even as production complexity increases, the system's logic remains manageable. By modeling complex events, such as equipment failures or resource shortages, UML statecharts help keep operations predictable and scalable across different manufacturing contexts. This flexibility and clarity in control logic make statecharts an indispensable tool for system designers tasked with managing intricate, asynchronous events [30].

2.3.4. Case Studies

The goal of this section is showing the application of UML to control policies, in particular it presents two case studies: the first focuses on the application of a release control policy using a system modeled with UML statecharts, while the second explores the implementation of scheduling policies through various UML diagrams.

2.3.4.1. CONWIP Control Policy and UML [29]

This chapter presents a use case that demonstrates the effectiveness of UML statecharts for modeling and managing control policies within manufacturing systems. The study titled "*Formal Modelling of Release Control Policies as a Plug-in for Performance Evaluation of Manufacturing Systems*", highlights how UML statecharts can provide a structured, adaptable approach to regulating part flow and managing system states through well-defined control policies. By implementing

statecharts within a modular, ontology-based framework, this approach illustrates the flexibility of UML statecharts in performance evaluation and control within manufacturing environments, particularly when CONWIP control policy is incorporated [29].

In manufacturing, control policies regulate the flow of materials and parts into workstations or production areas, significantly impacting lead time, work-in-progress (WIP) levels, and throughput. Modeling these policies effectively is challenging, particularly when high reconfigurability is needed. The use of UML statecharts enables modular representation of different control policies, making it possible to seamlessly switch or adjust policies based on evolving system requirements or performance insights [29].

The article underscores the potential of UML statecharts in modeling release control policies, particularly through the introduction of a modular, ontology-based framework that includes key manufacturing elements such as production resources, control policies, and production plans. This modular architecture employs UML state charts to represent release controllers as finite state machines, efficiently enabling transitions between states based on pre-defined conditions, thus offering a dynamic yet structured method for managing workflow decisions. UML state charts are instrumental in this application as they allow clear, rule-based transitions within manufacturing processes, which are represented as discrete states and transitions for each production resource, such as workstations and buffers. This not only provides flexibility but also supports the automatic generation of DES models, further demonstrating the effectiveness of UML state charts for modeling complex manufacturing scenarios [29].

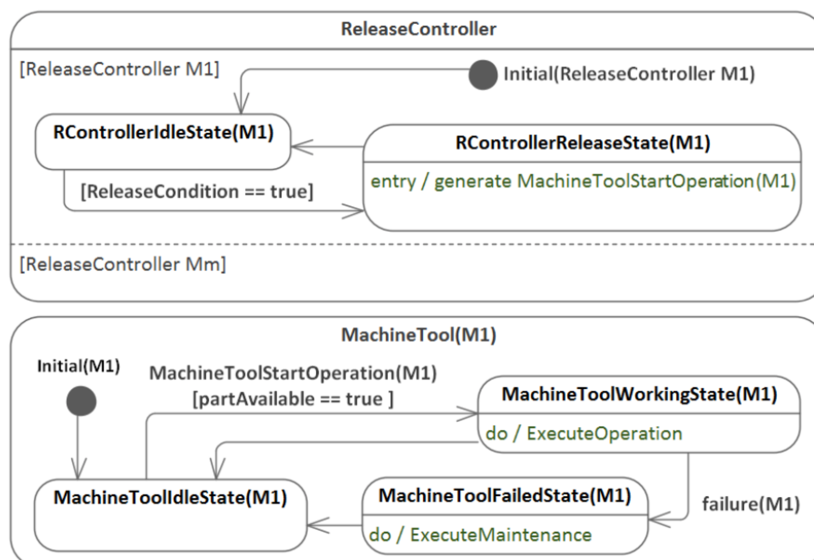

Figure 12: UML statecharts of Release Controller and Machine Tool [29]

The figure 12 shows two different statecharts, one for the controller, the other for the machine tool. The Release Controller starts in an idle state, waiting for a *ReleaseCondition* to be met. When true, it shifts to a release state, generating a *MachineToolStartOperation(M1)* event that signals Machine Tool M1 to begin work. The Machine Tool then moves from its idle state to a working state, executing its operation if a part is available. If a failure occurs, it transitions to a failed state for maintenance before returning to idle. This coordinated control system efficiently manages part flow and machine readiness, enabling responsive, structured manufacturing operations.

In applying this framework to a real-world scenario, a drawer slide assembly line in the furniture market, the authors validate the capability of UML state charts to manage both simple and complex control policies within a modular system architecture. The article illustrates how the modeling approach, supported by UML state charts, is able to integrate various control policies, such as the CONWIP (Constant Work-In-Process) policy, effectively optimizing throughput, work-in-progress (WIP), and resource utilization. This case study provides a compelling example of UML state charts being deployed not merely as a descriptive tool but as an actionable framework that supports both the design and evaluation phases of manufacturing systems, reinforcing the relevance of UML state charts as an adaptable, powerful instrument for managing and optimizing industrial processes within the realm of modern manufacturing. Thus, this work serves as a foundational case demonstrating the utility of UML state charts for detailed control policy modeling in manufacturing systems, justifying their selection as a versatile tool in the overarching thesis context [29].

In conclusion, this use case illustrates the effective application of UML state charts for modeling and managing release control policies within a manufacturing system. By structuring the Release Controller and Machine Tool as modular, condition-driven state machines, this approach allows for flexible and efficient control over production processes. The UML state charts enable clear transitions based on specific conditions and events, such as part availability or system failures, facilitating real-time responsiveness and coordination between system components. This structured framework not only enhances the system's adaptability to changing operational needs but also simplifies the integration of complex control policies, such as those needed for production flow optimization. As demonstrated, the UML state charts provide a powerful tool for achieving modular, scalable, and transparent control in automated manufacturing environments, making them highly suitable for industrial applications that require precision and flexibility.

2.3.4.2. Scheduling policy and UML

In agile production systems, the Agile[5] Production Planning and Control System (APPCS), developed by Tsai and Sato, exemplifies how UML statecharts can effectively model complex, real-time production environments with high variability and demand unpredictability. This system is tailored for make-to-order industries, where frequent changes in customer orders and supplier constraints demand a flexible system that can adapt swiftly to new conditions while maintaining a feasible production plan. The APPCS employs UML to establish a structured, responsive approach that enables the system to adapt dynamically to fluctuations in demand and supply, providing efficient, reliable production management [31].

At the heart of APPCS is its use of UML statecharts to represent changes in a job's lifecycle, detailing each phase from creation to completion. This state-based model offers a granular view of each job’s status, accommodating phases like planning, scheduling, rescheduling, and finalization. Within the UML framework, each job state, along with transitions driven by events, dependencies, and resource constraints, captures the complete job lifecycle. This allows APPCS to adjust swiftly in response to shifts in production demands, offering a real-time view of each job's progression and interdependencies between jobs, resources, and processes. This level of detail enhances the system’s responsiveness to evolving conditions in production workflows [31].

The APPCS model’s use of UML statecharts also includes specifications for behavioral transitions to manage supply disruptions or demand shifts. This agile configuration allows customer orders or supplier updates to trigger immediate adjustments in planning and scheduling, based on predefined transitions in the UML model. For example, if a supply delay occurs, the statechart directs the system to reallocate resources, adjust schedules, and reinitialize planning for affected jobs, ensuring that ongoing shop floor operations remain stable. This adaptability maximizes resource utilization, reduces waste, and enhances service performance, underscoring UML’s role in promoting operational flexibility and control precision [31].

In addition to statecharts, APPCS uses a class diagram to define core elements like parts, operations, work centers, and resources, establishing a static foundation for the state-based interactions. Statecharts, together with class and sequence diagrams, structure data relationships and dependencies among production resources, creating a clear framework for both static and dynamic system states.

[5] The Agile methodology is a project management approach that involves breaking the project into phases and emphasizes continuous collaboration and improvement [40].

For instance, each class attribute interacts according to the UML-defined state transitions, facilitating a seamless flow from demand generation through scheduling and production control. This design enables systematic scheduling based on the real-time status of each resource [31].

To further enhance adaptability, APPCS employs a behavioral model with specific rules for planning, scheduling, procurement, and control, embedded within the UML statecharts. These rules allow the system to distinguish between normal operations and exceptions, like resource shortages or customer-driven changes. Through state-dependent decision-making, APPCS dynamically adjusts production plans in response to resource availability, job priority, and external constraints, maintaining efficient production even in highly variable environments. This approach underscores UML's potential to support real-time, adaptive decision-making [31].

A visual dimension in APPCS comes from integrating Gantt charts, derived from UML sequence and state diagrams, to represent job sequences and resource allocations over time. These Gantt charts provide production managers with a tool to track progress, identify dependencies, and anticipate bottlenecks. Consequently, the UML model not only serves as a digital framework for APPCS but also offers a practical interface for real-time monitoring and adjustment, aligning well with agile production practices. This visual component further demonstrates the versatility of UML statecharts in translating complex schedules into actionable insights, enhancing control and adaptability in production planning [31].

In conclusion, APPCS showcases the utility of UML statecharts in managing dynamic production processes. This case highlights how UML statecharts facilitate real-time adaptability by modeling job lifecycles and transitions, enabling immediate response to demand, supply, and resource changes, capabilities crucial for make-to-order industries with high variability. UML's integration of statecharts, class diagrams, and sequence diagrams enables APPCS to represent static and dynamic system elements, demonstrating its applicability beyond software to operational fields where agility and rapid reconfiguration are essential. This use case validates UML statecharts as a highly effective tool for adaptive, real-time production management [31].

3. Methodology

The thesis work followed a structured and iterative process, comprising several key stages. The first phase involved a thorough analysis and understanding of the baseline system configuration, with a specific focus on the statecharts that were originally used to model the behavior of the simulated entities. A deep comprehension of these statecharts was crucial, as they provided the primary framework for capturing the system's dynamic behaviors, including events, states, and transitions.

After this initial assessment, the focus shifted to a detailed evaluation of the requirements for the new simulation model. This phase entailed a comprehensive review of the objectives and functional needs of the updated system, highlighting the differences from the initial framework. It became clear that, while the existing statecharts were effective in a system governed solely by release policies, they exhibited several shortcomings when applied to the expanded scope of the new simulation, which also incorporated dispatching policies. These limitations emphasized the need for modifications or, in some cases, a complete redefinition of the statechart representations to better reflect the more intricate behaviors required by the updated system.

In response to these challenges, the next phase centered on a systematic redesign of the statecharts. This process was driven by the necessity to incorporate the specific behaviors and interactions dictated by the new simulation's requirements. The redesign involved not only minor adjustments but also the creation of entirely new state representations, transitions, and event management structures.

Once the statecharts were defined, the implementation phase began. A Python simulation was developed to model the statechart behaviour across various scenarios and inputs. The system simulated included a buffer, a machine processing two distinct part types downstream of the buffer, and a controller executing specific control policies.

Following the completion of the redesigned statecharts and simulation model, the project progressed to the validation phase. Rigorous testing was conducted to ensure the accuracy and reliability of the new statechart configurations within the simulation environment. This phase involved evaluating the model's ability to replicate the intended behaviors and to respond accurately to a variety of inputs and scenarios. Validation was performed by comparing the Python simulation against Siemens Plant Simulation, using identical inputs and conditions to ensure the behavior of the objects in both simulations was consistent.

Through this structured and iterative approach, a robust set of use cases was developed, applying three different dispatching policies. Ultimately, the new model successfully addressed the limitations of the initial statecharts and demonstrated the ability to simulate the complex behaviors required by the enhanced system, providing a strong foundation for further analysis and experimentation.

3.1. Initial Context

This work began with the Virtual Learning Factory Toolkit (VLFT[6]), an advanced platform designed for modeling, simulating, and optimizing manufacturing systems through the use of digital twins and cutting-edge technologies. Developed by W. Terkaj and M. Urgo, the VLFT integrates virtual and augmented reality tools, enhancing both educational applications and the optimization of production systems. The platform allows for the creation of virtual replicas of workstations, machinery, and human resources, enabling real-time simulation and analysis [32].

Within this environment, a use case was utilized to provide a detailed exploration of a complex production line, highlighting the dynamic interaction between multiple stations and material buffers. The production line taken ad reference consists of several stations, each responsible for specific operations on the workpieces to produce a final product. Positioned between these stations are buffers, which temporarily store parts and regulate material flow. A distinctive feature of this case is the non-linear nature of the production line, which includes two parallel lines that converge in a buffer. This merging point requires precise coordination to ensure that parts from both lines are efficiently managed within the buffer, preventing bottlenecks and maintaining a smooth production flow throughout the system.

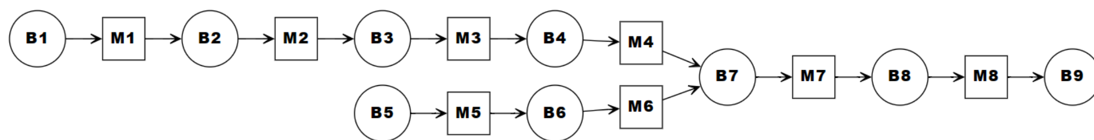

Figure 13: Layout of the line

The line structure, illustrated in Figure 13, shows how the two parallel lines converge, emphasizing the importance of release policies to control material flow from one station to the next, and the role of dispatching policies in deciding which operation should begin at any given time. In this system,

[6] The Virtual Learning Factory Toolkit (VLFT) is a set of existing digital tools to support advanced engineering education in manufacturing [32].

only release policies have been implemented, regulating when parts should move from the buffers to the next station, ensuring an optimal flow and preventing any station from becoming either starved or overloaded with workpieces. This balance between stations helps maintain the overall efficiency of the production process.

The case also addresses the challenge of managing the flow of different parts arriving from the two parallel lines. When parts from both lines converge at buffer "$B7$," careful control is needed to prevent congestion, which could otherwise disrupt the entire production line. Dispatching control policies become essential at this point, determining which part should proceed to ensure the downstream process remains smooth and free from delays.

This use case not only illustrates the complexity of managing multi-line production systems but also underscores the critical role of intelligent control mechanisms, such as release policies, in maintaining balance and efficiency throughout the production line. It reflects the types of challenges that modern manufacturing systems face, where optimizing material flow and preventing bottlenecks are key to ensuring high productivity and minimizing delays in the production cycle. Through this study, valuable insights can be gained into how advanced control strategies can be applied to complex manufacturing environments to enhance both performance and throughput.

3.1.1. Initial Statecharts

The initial simulation focused on implementing a release control policy and evaluating the overall performance of the production line. To achieve this, statecharts were designed to integrate release control logic directly into the behavior of the system's objects. This approach involved creating individual statecharts for each object, along with a dedicated controller statechart responsible for managing the release control policy. The controller regulated the flow of materials, determining when and how resources were introduced into the system to maintain smooth operations. By decoupling the control mechanism from the specific behaviors of individual objects, the simulation enabled flexible testing of different release policies, allowing for comprehensive performance evaluations and optimization opportunities.

A key feature of these statecharts is their modular design, enabling them to be applied across various systems with different characteristics while maintaining functionality. In the virtual environment, multiple systems with unique attributes were modeled using the same modular statecharts, demonstrating their adaptability and versatility. This modularity forms the foundation of this thesis,

with the primary goal being to create new modular statecharts that can be applied to entirely different systems, yet operate consistently and effectively across diverse environments.

- **Machine statechart:**

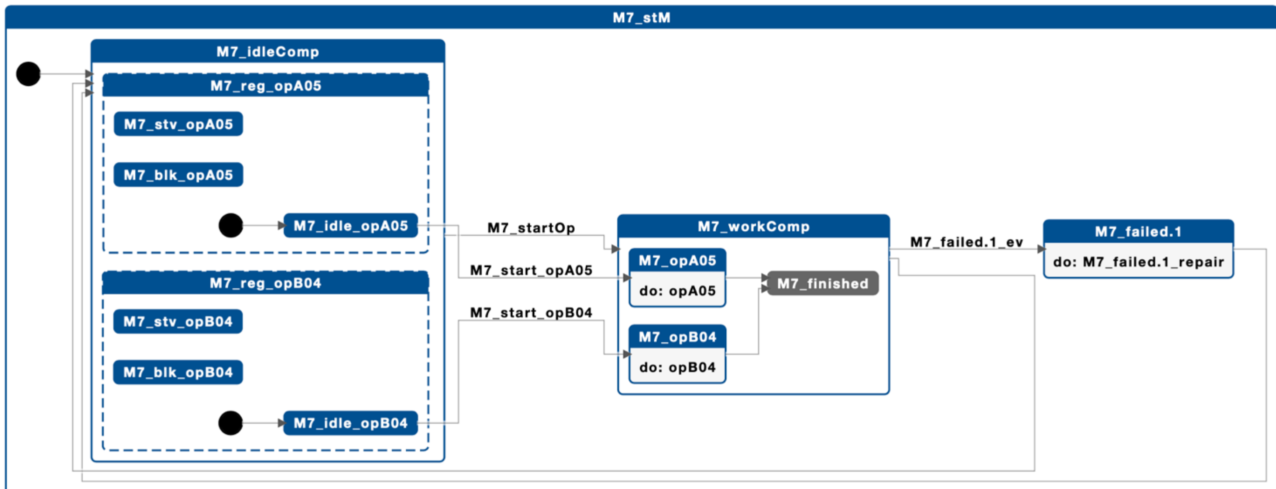

Figure 14: Machine (M7) statechart

The statechart illustrates the operational lifecycle of machine *M7*, starting with its idle and working phases and extending through possible failure and repair states. At the top level, the *M7_stM* state encapsulates the machine's core processes. Initially, the machine begins in the *M7_idleComp* state, where it can experience different conditions, such as being idle, blocked, or starved. These conditions are managed in an orthogonal state, allowing for the simultaneous tracking of both part types that the machine handles. This structure provides a detailed view of *M7*'s behavior across various operational scenarios, ensuring that all possible states of the machine are accounted for in the simulation. Once activated, the machine transitions to the *M7_workComp state*, where it performs the operations *opA05* and *opB04*. Each operation within the working state runs until completion, at which point the system transitions based on triggers such as the completion signal *M7_finished*. The chart captures the interrelation of operations as the machine alternates between work on different tasks.

However, the machine is also susceptible to breakdowns. If a failure occurs, triggered by the event *M7_failed.1_ev*, the machine moves into the *M7_failed.1* state, representing a malfunction. In this failure state, the machine undergoes a repair process, represented by the action *M7_failed.1_repair*, until it can return to operational status. The statechart effectively models the machine's movement between idle, operational, failure states, highlighting the different conditions under which it operates, and the transitions dictated by events like completion or failure triggers.

This statechart provides a structured view of the machine’s dynamic behavior and the critical states it may encounter during its functioning.

- **Controller statechart:**

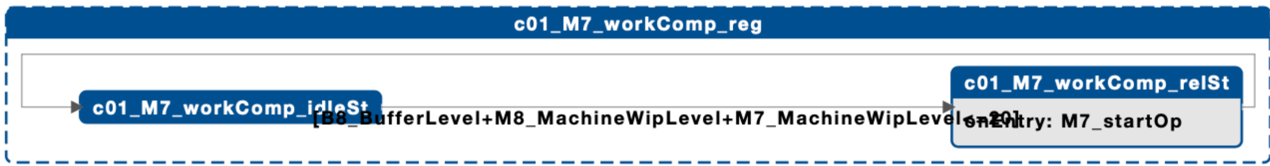

Figure 15: Controller (M7) statechart

This statechart represents a controller responsible for managing the operational flow and status of multiple machines, determining when each machine can begin work or must remain idle. The controller's primary function is to regulate machine operations by transitioning between idle and work states based on system conditions and release policies. For example, when the controller for a specific machine, such as *M7*, is in the idle state, it means the machine is not permitted to begin its operation, typically due to the constraints imposed by a release policy. In figure 15, the release policy is based on CONWIP (Constant Work-In-Progress), ensuring that only a specific number of jobs are active within a subset of the system at any given time.

When the controller transitions to the work state, it signals that the necessary guard condition on the transition have been met, allowing the machine to commence its assigned tasks. This indicates that resources, material availability, or buffer levels have aligned with the release policy, thus authorizing the machine to proceed with its operations. The connection between the controller and the machine’s statechart is managed through events. Once the release policy is verified by the controller, an event is sent to the machine. If the machine is in a state that allows it to start a new operation, it will proceed with its task.

Although each machine's controller operates independently, they interact within the larger system, evaluating conditions and coordinating with other system elements to ensure that the release policy is applied properly. This helps prevent machines from beginning operations prematurely or without sufficient resources, maintaining the balance and efficiency of the entire production line. The controller's role is critical in synchronizing machine activities, ensuring that workflows progress without bottlenecks or delays.

Moreover, the statechart demonstrates that these controllers are interconnected within the production system, ensuring that machines only start working when it is optimal for the system as a whole. This careful coordination helps maintain productivity, prevent bottlenecks, and manage the flow of materials between machines, contributing to the smooth and efficient functioning of the production line. The modular design of this controller provides flexibility, allowing each machine to be controlled individually while still adhering to a centralized policy that governs the overall system's operation.

3.2. New statecharts development

Building on the initial statecharts, the objective was to incorporate dispatching policies alongside the existing release policies while preserving the modularity of the statecharts, ensuring their applicability across different environments. A key feature maintained from the outset was also the communication between statecharts through events, allowing them to send signals and interact with one another.

To achieve this, modifications were necessary in the controller statechart, which initially only checked whether a machine could start working but did not define which specific part should be processed next. Implementing dispatching policies required updating the controller to not only manage the initiation of operations but also handle the selection and prioritization of parts. This enhancement addressed the missing functionality, enabling the model to effectively balance part processing decisions and optimize the overall system performance.

The first statechart to be modified was the controller statechart, with several assumptions made to ensure proper functionality under the new conditions. First, it was assumed that there is a single buffer upstream of the machine. The machine is capable of producing different part types, with each type corresponding to a specific operation. The buffer consists of positions, and the dispatching policy selects the part to be released based on the policy applied. The part's position in the buffer is linked to its order of arrival, ranging from 1 to the buffer's capacity. These assumptions guided the changes needed to implement effective dispatching in the system.

The focus of the analysis is on the subsystem comprising *B7, M7*, and the controller, as this is the critical point in the use case where the two separate production lines merge into a single line at *B7*. This convergence creates a complex flow that requires precise control, making it essential to understand how the controller manages the release and dispatching of parts at this junction, ensuring the seamless integration of both lines into the unified production process.

3.2.1. New Controller Statechart

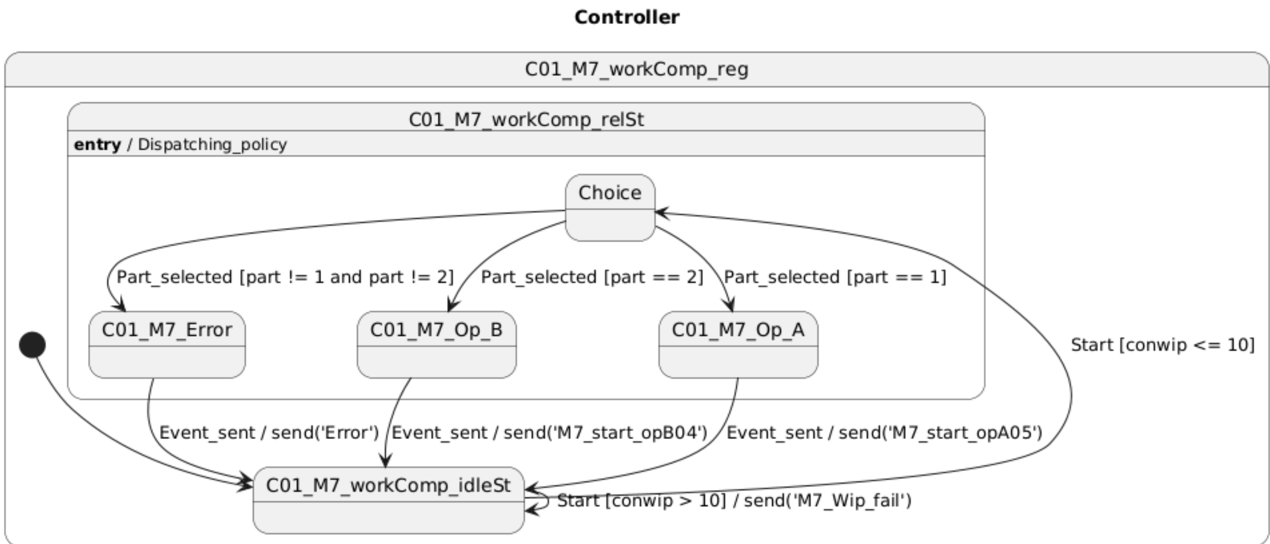

Figure 16: New Controller (M7) statechart [7]

The newly enhanced controller statechart now integrates both release and dispatching policies, significantly improving the system's decision-making capabilities. While the overall structure remains familiar, with the machine in an idle state until the release policy is met (managed as a guard condition on the transition), the introduction of dispatching adds a layer of control over the workflow.

Once the release policy is satisfied (such as meeting a CONWIP condition), the controller activates the dispatching policy on entry in the work state to determine which specific part type should be processed next. The system then transitions into a working state that now includes a choice element, a critical addition. This element forms the core of the dispatching policy, allowing for multiple possible transitions. Each transition is governed by a guard condition, and only the one that meets the guard criteria is activated, sending the appropriate event to the statechart to trigger the desired activity.

When the dispatching policy is activated upon entering the work state of the controller, it outputs the part type to be processed and, thanks to the guards the statechart directs the controller to the chosen operational states. For instance, if part type 1 is selected, the statechart transitions to the *C01_M7_Op_A* state, sending the event *M7_start_opA05* to the machine to initiate operation A. Similarly, if part type 2 is chosen, the system moves to *C01_M7_Op_B*, sending the event *M7_start_opB04* to trigger operation B on the machine.

[7] Statechart automatically generated using *PlantUML*

Additionally, the statechart includes an error management mechanism through the *C01_M7_Error* state. This state is triggered if the dispatching algorithm select an invalid part type or if no available parts are present for processing. In such cases, the controller enters the error state and sends an "*Error*" signal, preventing incorrect instructions from being sent to the machine. The system then resets and returns to the idle state, ready for the next another part selection.

This enhanced structure not only ensures more precise control over the flow of work but also provides robust error handling, making the system more resilient and capable of handling complex production scenarios efficiently.

The enhanced statechart offers a more robust and dynamic approach to machine control. By incorporating both release and dispatching policies, the controller can better manage the flow of parts through the system, ensuring that each machine only processes the correct part at the appropriate time. Furthermore, the inclusion of error handling ensures that the system remains reliable and avoids processing issues. This modular and scalable design allows the system to optimize production flow while maintaining flexibility in handling various part types and operational conditions.

3.2.2. New Machine Statechart

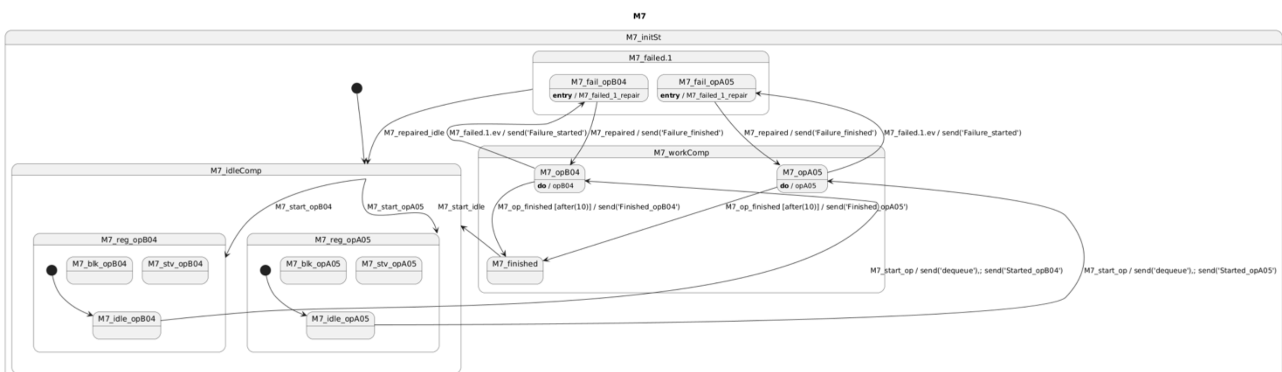

Figure 17: New Machine (M7) statechart [8]

The updated machine statechart incorporates several important refinements to improve the system's functionality, especially in managing failures. One key modification is the introduction of specific failure substates corresponding to each working substate. This ensures that, in the event of a failure, the part type being processed is retained, so no information is lost about the operation in progress when the failure occurs.

[8] Statechart automatically generated using *PlantUML*

Additionally, the statechart now features event-triggered notifications at various stages of operation. Each time a new operation begins, the statechart sends an event detailing the part type being processed (either *opB04* or *opA05*), ensuring that the system can track exactly what is happening at any given moment. Similarly, when an operation is completed, another event is sent, signaling the successful completion of that specific task.

This event-driven structure also applies to failure handling. When a failure starts, an event is sent out, and upon completion of the repair process, another event is dispatched to notify the system that the machine has been successfully repaired. These events are crucial for maintaining real-time visibility of the machine's status throughout its lifecycle.

The failure states are directly tied to their corresponding operational states, meaning that if a failure occurs during '*opA05'* or '*opB04'*, the machine transitions into the respective failure substate. Once the failure is resolved, the machine moves back to its operational state, resuming the processing of the part without losing any context.

In this specific use case, a guard condition is applied to the transition from the working activity to *M7_finished*, incorporating a time constraint. This condition requires a duration of 10 seconds in the working state before the transition to *M7_finished* is triggered, ensuring that the system only advances after the specified time has elapsed.

In summary, the improved statechart enhances both the traceability and resilience of the machine's operations by ensuring that all transitions, whether they involve normal workflow or failure conditions, are explicitly tracked and managed. This not only aids in monitoring the system's performance but also helps in debugging and understanding the machine's behavior in real-time.

3.2.3. Buffer Statechart

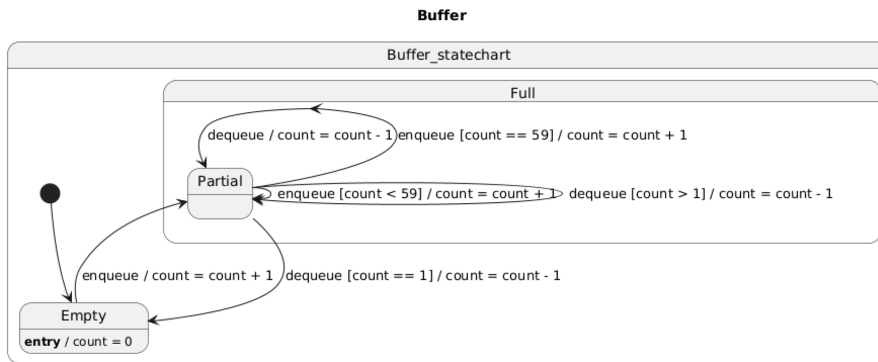

Figure 18: New Buffer (B7) statechart [9]

To complete the subsystem design, the Buffer statechart was developed, modeling the dynamic behavior of a buffer as it transitions between three states: *Empty*, *Partial*, and *Full*. For this use case, the buffer's capacity is set to 60 items. It begins in the Empty state, where no items are held, and the count is initialized at zero. When an enqueue operation occurs (an item is added), the count increases, moving the buffer into the Partial state, where the item count ranges from 1 to 59. In this state, both enqueue and dequeue operations are allowed: enqueue increases the count, while dequeue reduces it. If the count reaches 60, the buffer transitions to the *Full* state.

In the *Full* state, the buffer has reached its maximum capacity, so only dequeue operations are allowed, which reduce the count. If the buffer empties to a count below 60, it transitions back to the *Partial* state. If continued dequeuing reduces the count to 1, the buffer moves back to the *Empty* state, resetting the count to zero.

Each state in the buffer system has defined rules governing *enqueue* and *dequeue* actions. The *Empty* state allows only *enqueue* actions, while the *Full* state only permits *dequeue* operations to prevent overfilling. The *Partial* state, the most flexible, allows both actions based on the current count. This ensures that the buffer functions within its defined capacity, avoiding overflows or underflows, and maintaining proper order and flow of items.

The statechart actively tracks buffer levels, ensuring that system constraints are respected. State transitions occur as the buffer's capacity changes, providing real-time tracking and control over

[9] Statechart automatically generated using *PlantUML*

system behavior. This careful management ensures predictable, efficient buffer operations, regardless of whether it is in an empty, partially filled, or full condition.

In conclusion, the three statecharts allows for the integration of both release and dispatching policies significantly enhancing the flexibility and control of the system. The modifications made to the controller statechart allowed for precise part selection and prioritization, improving the system's responsiveness to different conditions. By incorporating event-triggered notifications and error-handling mechanisms, the statecharts for both the controller and machine enabled better traceability and real-time visibility. Finally, the buffer statechart completed the subsystem, ensuring that material flow was efficiently managed within the system’s constraints, contributing to a seamless and optimized production process.

4. Implementation

After defining the statecharts, the analysis proceeded by simulating their behaviour under various conditions to assess whether the system's performance aligned with expectations. This step was crucial for evaluating the statecharts' ability to handle different operational scenarios and verifying that their behaviour matched the intended design. Through simulation, the system's dynamic responses were tested to ensure consistency, accuracy, and functionality, allowing for a comprehensive validation of the statechart models before further development. The programming language used for the simulation is Python, specifically leveraging the Sismic library developed by Alexandre Decan.

4.1. Introduction to Sismic library

The Sismic Python library, created by Alexandre Decan, is designed to execute, simulate, and validate statecharts. It offers a comprehensive framework for defining and managing statecharts by incorporating advanced methodologies like Test-Driven Development (TDD[10]), Behavior-Driven Development (BDD[11]), and Design by Contract (DbC[12]). Unlike traditional tools focused on visualization and simulation, Sismic extends its capabilities with robust testing and runtime verification, ensuring that systems behave correctly under various conditions and automatically verifying compliance with predefined contracts or behavioral properties [33].

4.1.1. Key Features of Sismic

Sismic offers a robust set of features for statechart execution and testing. It allows users to define statecharts in YAML[13], a human-readable format, making it easy to build and maintain complex

---

[10] Test-driven development (TDD) is a way of writing code that involves writing an automated unit-level test case that fails, then writing just enough code to make the test pass, then refactoring both the test code and the production code, then repeating with another new test case [41].

[11] Behavior-driven development (BDD) is an Agile software development methodology in which an application is documented and designed around the behavior a user expects to experience when interacting with it [42].

[12] *Design by contract (DBC)* is a method originally proposed by Betrand Meyer [97] for the design of object-oriented and component-oriented systems. The main characteristic of DBC is that classes define their behavior and interplay by *contracts [43].*

[13] YAML is a human-friendly data serialization language for all programming languages [44].

models. Sismic integrates with PlantUML[14] to visualize statechart structures, providing a clear representation of system behavior. At its core, Sismic's interpreter follows run-to-completion semantics, processing one event fully before the next, while also supporting timed events, critical for real-time systems [33].

Python integration allows users to embed custom Python code directly into statechart actions and guards, enabling advanced logic and seamless interaction with other Python applications. This feature extends statechart functionality and provides flexible control over behavior.

One of Sismic's most powerful features is its runtime verification through Design by Contract (DbC). Users can define preconditions, postconditions, and invariants directly within the statechart. During execution, these contracts ensure that the system's behavior adheres to specified rules, making it especially useful for high-reliability systems. Sismic also supports inter-statechart communication, which allows for the simulation of complex, parallel-running processes that must coordinate their activities [33].

For testing, Sismic integrates Test-Driven Development (TDD) and Behavior-Driven Development (BDD). Unit tests can be written for individual statecharts, ensuring that components behave as expected. BDD allows domain experts to define test scenarios in natural language, which Sismic maps to the statechart for execution, facilitating collaboration with non-technical stakeholders [33].

Lastly, Sismic is equipped to handle and simulate error conditions. It enables the testing of failure states within the system, ensuring that the system responds predictably under fault conditions, enhancing system reliability and robustness [33].

Sismic's architecture is modular and designed for extensibility. It consists of several components: *sismic.model* API for creating and manipulating statecharts directly, *sismic.io* API for importing/exporting statecharts and integrating with external visual tools like PlantUML, *sismic.interpreter* which manages the execution of statecharts, handles guards, and processes actions. Sismic is easily extendable, allowing developers to customize its behavior, add new action languages, or integrate it with larger systems [33].

[14] PlantUML is a highly versatile tool that facilitates the rapid and straightforward creation of a wide array of diagrams [45].

4.1.1.1.Statechart definition

In Sismic, statecharts are defined in YAML, a structured yet readable format that enables detailed modeling of complex systems. The YAML format organizes the components of a statechart into elements such as states, transitions, events, actions, and guards.

States are defined hierarchically, supporting composite, orthogonal, and parallel states. Each state may have entry and exit actions and can contain initial and history states. The transitions specify triggers and optional conditions (guards) that must be met for the transition to occur. Each transition can also define actions that execute when the transition is triggered. Guards can be used to control transitions based on logical conditions, making state behavior more dynamic and adaptable.

Additionally, events are defined to trigger transitions between states. Sismic also handles timed events, allowing for event-driven modeling in real-time systems. History states can be employed to remember the last active substate, which ensures that, when returning to a composite state, the system can restore the previous state instead of starting from the default initial state.

The YAML format enables concise, clear definition of all these components, making it easier for developers to model, test, and simulate statecharts while integrating smoothly with the Python ecosystem. Overall, Sismic's YAML format allows for highly flexible and detailed modeling of reactive systems [34].

4.1.1.2.Statechart visualization

Sismic provides visualization capabilities to represent statecharts in graphical form, facilitating better analysis and understanding of complex models. Statecharts can be visualized by exporting them to PlantUML, a widely-used tool for generating UML diagrams. Sismic supports exporting in both PNG and SVG formats, enabling clear, scalable images that capture the entire structure of the statechart. This functionality aids in identifying states, transitions, and their interactions more clearly, making it easier to track behavior, debug, and present the system's logic visually [34].

4.1.1.3.Statechart execution

Sismic's execution model is designed to simulate and control the execution of statecharts through an interpreter. This interpreter processes events in a step-by-step manner, adhering to a run-to-completion approach. When an event is triggered, the system completes all necessary transitions and

actions before handling the next event. The interpreter also supports timed events, which are critical for real-time systems, allowing users to specify delays or periodic triggers.

Sismic allows users to manually queue events for execution and monitors the statechart's progress through logging and inspection. It also supports real-time execution and simulated time, offering flexibility in testing scenarios. Users can define specific time constraints and delays, which the interpreter handles through a simulated clock. This is particularly useful for testing systems that have time-sensitive behaviors, such as alarms or timed workflows.

Sismic also supports concurrent and parallel states, which means that it can manage statecharts with multiple active states at once, allowing users to model and execute more complex systems. When handling parallel states, the interpreter executes events and transitions in different regions simultaneously, ensuring that all active regions progress without delay or conflict.

The execution flow can be paused, resumed, or restarted, offering full control over the simulation. During execution, the current configuration of active states is accessible, and Sismic provides information about events processed, actions performed, and transitions taken, which facilitates detailed inspection and debugging.

In addition, Sismic integrates runtime verification, which enables users to monitor the statechart's execution against specified contracts or conditions. This ensures that the system's behavior adheres to defined rules during its runtime, alerting users to violations in real-time.

The process of defining, running, and executing statecharts in Sismic is highly efficient, especially due to its ability to support concurrent execution of multiple statecharts. Each statechart can operate in parallel through orthogonal regions, which act as independent units within the system. These regions can process events independently, handle transitions, and perform actions, while remaining in sync with other regions through shared events. This capability allows Sismic to model complex systems that require real-time interaction and synchronization between different components, such as distributed or multi-threaded systems.

The ability to run concurrent statecharts is particularly valuable for simulating systems where multiple processes need to operate in parallel while interacting with one another. For instance, in an industrial automation setting, different machines can be modeled as independent statecharts that run simultaneously but communicate through shared events, such as a signal indicating part

availability or system-wide alarms. These statecharts, running concurrently, ensure that each machine responds to system triggers in a synchronized manner.

Furthermore, shared events in Sismic provide seamless coordination between the concurrent statecharts. When one statechart triggers an event, it can impact the transitions or actions of another, ensuring that different subsystems react cohesively. This shared event-driven model ensures that statecharts, even when operating independently, remain part of a coordinated system, responding to events in a synchronized manner. This makes Sismic particularly effective for simulating complex, real-time systems where multiple processes need to work in tandem, such as in telecommunications or networked systems [34].

The Sismic library include also methods for handling time constraints in statecharts through internal clocks, time-based transitions, and event scheduling. It describes how to implement time predicates like *after(x)* (for triggering a transition after a specified time in a state) and *idle(x)* (for triggering when idle for a set duration). The library allow three types of clocks:

- *SimulatedClock*: Allows control over time flow in simulations, manually or in real-time.
- *UtcClock*: Syncs with real-world time, suitable for applications needing actual-time alignment.
- *SynchronizedClock*: Ensures synchronized operation of multiple statecharts in coordinated workflows.

Time-related events can be delayed triggering at precise times, even allowing for chaining multiple timed events. Each clock type supports automated transitions and controlled event sequences, useful for both simulated environments and real-world applications needing precise time alignment.

In summary, Sismic is a robust, open-source Python library that enhances statechart modeling with advanced features for testing, simulation, and runtime verification. By supporting techniques like Test-Driven Development (TDD), Behavior-Driven Development (BDD), and Design by Contract (DbC), Sismic bridges the gap between traditional statechart modeling and reliable system behavior. Its ability to manage concurrent statecharts and facilitate event-driven synchronization makes it invaluable for modeling complex, event-driven systems requiring high levels of coordination, precision, and parallelism. This makes it highly beneficial for both researchers and industry professionals across various fields.

4.2. Hypothesis of the simulation

The Python simulation developed for this work has been designed with the goal demonstrating the correctness of the statecharts of modeling a production system comprising a single buffer, a downstream machine that processes two distinct part types, and a controller that manages both release and dispatching policies. The goal is to observe the system's behavior under various dispatching and release approaches. Three specific dispatching policies: FIFO (First In, First Out), LIFO (Last In, First Out), and Earliest Due Date are employed, along with a CONWIP (Constant Work In Progress) release policy, allowing for detailed comparison of different workflow dynamics.

An important feature of this simulation is its fixed duration, structured so that all statecharts operate within a defined timeframe. A simulated clock is activated by each statechart, tracking each time instant. This clock advances by one second upon completion of the events queue, ensuring that each statechart is aligned to process the full queue every second. This approach is crucial as the machine statechart's working activity is bound by a fixed duration, during which state transitions are restricted, maintaining consistency in the machine's operational cycle.

The simulation requires two main key inputs:

- **Parts arrival times**:

Key inputs for the simulation include the arrival times of parts to the buffer. When a part arrives to an empty buffer, both the buffer and the machine immediately begin processing that part without delay. If the buffer not empty, the part is added to the queue at the precise moment of arrival. Notably, if a part arrives at the same instant that a new work initiation occurs, the simulation prioritizes selecting a part for the machine and initiating processing before adding the new part to the buffer. However, all actions still occur within the same exact time instant, maintaining a seamless flow in operations.

- **Failure events**:

Failure events also play a critical role as inputs, and they can only occur while the machine is actively processing a part. If a failure occurs, the part currently being processed undergoes rework after the failure end, with processing restarting from the beginning. For instances of multiple failures affecting the same part, the operation will restart each time, continuing until the machine successfully completes an uninterrupted cycle.

Each simulation time step introduces a complete list of potential events to the statecharts governing the machine and the controller. If the machine or controller is in an appropriate state to process an event, it reads and executes it accordingly. Importantly, because the queue of events is processed only once for each working cycle, a brief one-second idle period is introduced between tasks. This idle time results from the structure of the event sequence, which ends with the completion of the current part.

The goal of the simulation is to evaluate the stability of the statechart system under various event queues, assessing its responsiveness to changes such as part arrivals and failure occurrences.

4.3. Simulation code

The simulation is organized as follows: first, it includes functions to implement the selected policies: FIFO, LIFO, Earliest Due Date (EDD), and CONWIP, each defining the rules for dispatching and release.

In the simulation, the FIFO function is a concise routine that selects the first item from the list of parts presents in the buffer at that instant of time, ensuring items are processed in the order they arrive:

```
def FIFO(jobs):
    if not jobs:
        return None
    return jobs[0]
```

Figure 19: FIFO Function of the simulation

Similarly, the LIFO function selects the last item from the buffer, ensuring the most recently added items are processed first:

```
def LIFO(jobs):
    if not jobs:
        return None
    return jobs[-1]
```

Figure 20: LIFO Function of the simulation

Lastly, the EDD function selects the item in the buffer with the closest due date by reading the due date attribute of each part, prioritizing those due soonest.":

```
def EDD(jobs):
    if not jobs:
        return None
    return min(jobs, key=lambda job: job['edd'])
```

Figure 21: EDD Function of the simulation

The CONWIP policy is implemented as a function that generates a random number within a specified range. Given that the system consists only of a buffer and a machine, it lacks a method to determine the actual CONWIP level. This random number, therefore, acts as a simplified proxy. If the generated

value falls outside an acceptable range, the statechart pauses and does not advance with the dispatching policy.

```
def conwip():
    return random.randint(1,10)
```

Figure 22: CONWIP Function of the simulation

Following these, the main code simulates the behavior of the statecharts representing each system component. These statecharts operate over a fixed time period, reacting to the sequence of input events, and execute on a per-second basis.

Throughout the simulation, the system tracks several key metrics. It logs each state entered by the statecharts, as well as any events triggered by them, providing a detailed record of system activity. Additionally, the buffer level is recorded at each time instant, capturing fluctuations in part inventory and offering insights into buffer dynamics and system flow across the entire simulation period.

The simulation operates on three statecharts: Buffer (B7), Machine (M7), and the Controller and these statecharts are loaded into the simulation using the following command:

```
buffer_statechart = import_from_yaml(filepath= buffer)
machine_statechart = import_from_yaml(filepath= machine)
controller_statechart = import_from_yaml(filepath=controller)
```

Figure 23: Statechart import command

Once the statecharts are loaded, an interpreter is created for each statechart to manage their execution. The interpreter processes events and ensures transitions occur as defined in the statechart. This is done using the following command:

```
buffer_interpreter = Interpreter(buffer_statechart, clock=clock)
machine_interpreter = Interpreter(machine_statechart, clock=clock)
controller_interpreter = Interpreter(controller_statechart, clock=clock)
```

Figure 24: Statechart interpreter command

To ensure smooth communication and coordination between the statecharts, Sismic's binding feature is employed. This feature allows the statecharts to send and receive events, facilitating their interaction. This synchronization is crucial for accurately simulating complex workflows, such as

when the machine starts working and the controller verifies all release and dispatching policies. The statecharts are interconnected using the following command:

```
controller_interpreter.bind(machine_interpreter)
machine_interpreter.bind(controller_interpreter)
machine_interpreter.bind(buffer_interpreter)
buffer_interpreter.bind(machine_interpreter)
```

Figure 25: Statechart bind command

This binding mechanism ensures that the statecharts function in a coordinated manner, enabling the controller to manage events like part processing and machine operation efficiently, while adhering to the defined system policies.

Once the code is executed, the simulation begins and runs for a fixed duration, which for this analysis has been set to 600 seconds. The simulation operates on the principle of providing, for each second of the simulation, a complete list of events to all three statecharts: Buffer, Machine, and Controller. This ensures that each statechart will react only to the events that are relevant based on its current state. The event queue used in the simulation is as follows:

```
queue = ['Start','Part_selected', 'Event_sent', 'M7_start_op', 'M7_op_finished', 'M7_start_idle']
```

Figure 26: Simulation queue

In this queue, the first three events (*'Start'*, *'Part_selected'*, and *'Event_sent'*) are designed to activate the controller, prompting it to verify policies and trigger necessary transitions. The last three events (*'M7_start_op'*, *'M7_op_finished'*, and *'M7_start_idle'*) are related to the machine's operation.

This method of sending a complete list of events each second ensures that all statecharts are given the opportunity to respond to any relevant events in real time. By doing so, the simulation ensures that no critical event is missed and that the statecharts can react to every possible input as the system evolves over time. This approach allows for precise, real-time coordination between the controller and the machine, reflecting the behavior of a real-world system with strict timing constraints.

i. Simulation Setup and Initialization

The simulation begins with the initialization of a simulated clock, which governs the timing of the entire simulation. This clock increments by one unit each time the events queue is completed,

managed through the variable *clock.time*. This structured increment ensures that each statechart aligns with the simulated time framework, allowing for consistent processing of events and transitions.

```
clock = SimulatedClock()
```

Figure 27: Clock initialization

The simulation starts by setting a fixed duration of 600 seconds:

```
while clock.time<=duration:
```

Figure 28: Simulation time constraint

Each second, the system evaluates the statecharts by reading inputs (events such as part arrivals and failures), which are processed by the interpreters associated with each component (buffer, machine, and controller).

At time = 0, the simulation begins by reading parts from the input part DataFrame, which records the arrival times and characteristics of each part. These parts are processed by the buffer statechart via an '*enqueue*' event.

```
if clock.time == 0:
    for part in input_part:
        if clock.time == part['arrival_time']:
            buffer_interpreter.queue('enqueue')
```

Figure 29: Sending enqueue event

The buffer interpreter processes the '*enqueue*' event, transitioning to the appropriate state based on buffer conditions (e.g., full or not). If the buffer is not full, the part is added to the buffer (jobs list), and the count of parts in the buffer is updated:

```
for step in buffer_interpreter.execute():
    for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
        if step.entered_states:
            b_state = step.entered_states
jobs.append({
        'part_type': part['part_type'],
        'arrival_time': part['arrival_time'],
        'part_name': part['part_name'],
        'edd': part['edd']
    })
count_buffer = buffer_interpreter.context.get('count', 'undefined')
```

Figure 30: Controller execution and simulation buffer update

At time = 0, the dispatching policy is applied to select the first job in the buffer.

```
if clock.time == 0:
    job = LIFO(jobs)
    part_type = job['part_type']
    part_chosen = job['part_name']
    controller_interpreter.context['part'] = part_type
```

Figure 31: Dispatching policy calculation

Once the part is chosen, the simulation proceeds by iterating through the event queue, which consists of six key events. These events trigger either the controller or machine statechart, enabling each to progress through its respective states. Thus, each queue iteration executes the three statecharts.

```
for q in queue:

    # Assignement to all the statechart the input queue
    machine_interpreter.queue(q)
    controller_interpreter.queue(q)
    buffer_interpreter.queue(q)
```

Figure 32: Queue for cycle initiation

ii. Controller

a. Conwip

The first event that activates the controller enforces the CONWIP (Constant Work in Progress) policy. This evaluation involves checking the WIP level generated by the simulation against the guard condition specified in the statechart. Specifically, the *conwip()* function calculates the current WIP level and updates the controller's context with this value. This ensures that the controller consistently operates in line with the CONWIP policy by allowing or restricting new work based on the established WIP threshold. If the policy is not verified, the statechart will block the transition to the work state. The policy is evaluated only once per time instant, so in cases where the CONWIP policy is not verified, an additional second will pass in the simulation. Consequently, if the machine is in an idle state, it will remain idle for an amount of extra second until the policy is verified.

```
wip = conwip()
controller_interpreter.context['conwip'] = wip
```

Figure 33: Conwip policy calculation

The WIP level determines how many parts can be released to the buffer, ensuring the system doesn’t overload and maintains a balanced workflow.

b. Dispatching Policy and Controller execution

The controller statechart is executed by iterating through its events. It reads the input queue and applies the appropriate dispatching policies to determine which part the machine should process next.

Following the selection of a part based on the specified dispatching policy, the relevant variable within the statechart is updated using the *'.context'* function.

```
for step_controller in controller_interpreter.execute():
    for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
        if step_controller.entered_states:
            c_state = step_controller.entered_states
    # Dispatching policy implementation
    if hasattr(step_controller.event, 'name') and step_controller.event.name == 'Start':
            part_type = 0

            if jobs:
                job = LIFO(jobs)
                part_type = job['part_type']
                part_chosen = job['part_name']

            controller_interpreter.context['part'] = part_type
            part_type = 0
```

Figure 34: Controller execution and dispatching policy calculation

iii. Machine

a. Execution

The machine statechart operates at each time instant, processing events related to part handling, such as initiating and completing operations. It transitions between states such as *working*, *idle*, or *failed* based on the events it receives. Upon starting an operation, the selected part is removed from the buffer, and the working activity commences. This activity is constrained by a guard *after(10)*, which ensures that each working phase lasts precisely 10 seconds according to the simulated clock, maintaining time consistency within the model.

```
for step_mach in machine_interpreter.execute():
    for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
        entered_states = step_mach.entered_states

    if any(isinstance(event, InternalEvent) and event.name == 'dequeue' for event in step_mach.sent_events):
        count_removed +=1
        remove = remove_from_queue(jobs)
```

Figure 35: Machine execution and remove from the queue called

When the machine statechart detects the internal event dequeue, it triggers a specific function. The purpose of this function is to remove the selected part from the simulation buffer, using the following commands:

```python
def remove_from_queue(jobs):
    if jobs:
        # Write the dispatching policy used in the simulation
        job_to_remove = LIFO(jobs)
        if job_to_remove:
            jobs.remove(job_to_remove)
        return job_to_remove
```

Figure 36: Remove from the queue function

b. Failure and Repair Management

The simulation tracks machine failures and repairs using a separate failure log (*df_failure*). If during processing time a failure is detected at a specific time (based on the log), the machine statechart transitions into a failure state. This happens sending the failure event and executing the statechart.

```python
if clock.time == np.floor(row['2']) and row['5'] == 1:
    machine_interpreter.queue('M7_failed.1.ev')
    for step_fail in machine_interpreter.execute():
        for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
            entered_states = step_fail.entered_states
            exited_states = step_fail.exited_states
```

Figure 37: Failure start event reading

The machine remains in the failure state until the repair event is triggered, prompting its transition back to the operational state. In this instance, the failure event is also drawn from the failure input list.

```python
if clock.time == np.floor(row['2']) and row['5'] == 0:
    machine_interpreter.queue('M7_repaired')
    for step_fail in machine_interpreter.execute():
        for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
            entered_states = step_fail.entered_states
            exited_states = step_fail.exited_states
```

Figure 38: Failure finish event reading

This allows the simulation to accurately model system disruptions and test the robustness of the dispatching and release policies under failure conditions.

Once the failure is resolved, as specified in the initial hypothesis, the affected part must undergo complete rework, necessitating that the operation be restarted and carried out from beginning to end without interruption. This rework process is managed automatically by the statechart.

iv. Buffer

a. Part arrival

For time steps greater than zero, parts arrive dynamically based on the values in the *input_part* data frame. The code checks if the current simulation time matches the part's arrival time and add the part to the buffer if it matches.

```
if clock.time != 0:
        for part in input_part:
            if clock.time == np.floor(part['arrival_time']):
                buffer_interpreter.queue('enqueue')
                for step in buffer_interpreter.execute():
                    for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
                        #write_to_file(output_file, '{}: {}'.format(attribute, getattr(step, attribute)))
                        if step.entered_states:
                            b_state = step.entered_states
                jobs.append({
                        'part_type': part['part_type'],
                        'arrival_time': part['arrival_time'],
                        'part_name': part['part_name'],
                        'edd': part['edd']
                    })
                count_buffer = buffer_interpreter.context.get('count', 'undefined')
```

Figure 39: Part arrival code

The buffer statechart controls the system's capacity, ensuring that new parts are only enqueued when there is available space. If the buffer is empty, parts are processed immediately upon arrival, as outlined in the earlier hypothesis. If the buffer is not empty the part arrival is processed at the end of the queue processing, ensuring that all incoming parts are appropriately managed regardless of the buffer's status. This approach guarantees that the system accurately simulates part arrivals and ensures seamless transitions between buffer states.

b. Buffer execution

The buffer is executed at each time instant, although it does not process queued events. Instead, *enqueue* events are triggered with each new part arrival, while *dequeue* events are sent directly from the machine whenever it begins processing a part. This setup enables the buffer to accurately track and update its part count in real-time. The buffer level is continuously monitored and stored in a variable by reading the count attribute from the statechart, which automatically updates with each *enqueue* and *dequeue* event. This approach ensures that the buffer's inventory is consistently and precisely recorded throughout the simulation.

```python
for step in buffer_interpreter.execute():
    for attribute in ['event', 'transitions', 'entered_states', 'exited_states', 'sent_events']:
        count_buffer = buffer_interpreter.context.get('count', 'undefined')
        if step.entered_states:
            b_state = step.entered_states
```

Figure 40: Buffer execution

In conclusion, this simulation effectively models the dynamic interactions between the buffer, machine, and controller using Sismic's statechart framework. By incorporating real-time event handling, dispatching policies, and failure management, it successfully replicates realistic manufacturing scenarios where components must operate in synchronization while adapting to changing conditions. The event-driven approach ensures that each statechart responds accurately to both expected events, such as part arrivals, and unexpected disruptions, like machine failures, providing a comprehensive view of system behavior.

The simulation starts by taking a series of input events and models the system's behavior, accordingly, allowing for an in-depth understanding of how the statecharts handle various scenarios. This approach evaluates whether the statecharts can effectively simulate a real-world environment under different conditions, making them suitable for integration into a complete digital twin. By demonstrating their ability to manage diverse manufacturing processes, these statecharts lay the foundation for a fully functional and reliable digital twin framework.

4.4. Simulation output

The output of the simulation is an Excel file that provides a detailed record of what occurred at each second of the simulation. This file serves as a comprehensive log, tracking key activities and state changes across the buffer, machine, and controller components.

Time	Controller_state	Machine_state	Buffer_state	Parts_in_buffer	Part_event	Fail_event	Part_chosen
21	['C01_M7_workComp_idleSt']	['M7_idleComp']	['Partial']	5	[InternalEvent('Finished_opB04')]	['X', 'X']	parttypeB_4
22	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_3
23	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
24	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
25	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
26	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
27	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
28	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	4	['X', 'X']	['X', 'X']	parttypeB_4
29	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	6	['X', 'X']	['X', 'X']	parttypeB_4
30	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	6	['X', 'X']	['X', 'X']	parttypeA_4
31	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	6	['X', 'X']	['X', 'X']	parttypeA_4
32	['C01_M7_workComp_idleSt']	['M7_idleComp']	['Partial']	6	[InternalEvent('Finished_opA05')]	['X', 'X']	parttypeA_4
33	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	5	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_4
34	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	5	['X', 'X']	['X', 'X']	parttypeB_5
35	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	5	['X', 'X']	['X', 'X']	parttypeB_5
36	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	5	['X', 'X']	['X', 'X']	parttypeB_5
37	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	5	['X', 'X']	['X', 'X']	parttypeB_5
38	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	5	['X', 'X']	[InternalEvent('Failure_started')]	parttypeB_5
39	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	6	['X', 'X']	['X', 'X']	parttypeB_5
40	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	6	['X', 'X']	['X', 'X']	parttypeA_5
41	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	7	['X', 'X']	['X', 'X']	parttypeA_5
42	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	7	['X', 'X']	['X', 'X']	parttypeB_6
43	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	7	['X', 'X']	['X', 'X']	parttypeB_6
44	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	7	['X', 'X']	['X', 'X']	parttypeB_6
45	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	8	['X', 'X']	['X', 'X']	parttypeB_6
46	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	8	['X', 'X']	['X', 'X']	parttypeB_7
47	['C01_M7_workComp_idleSt']	['M7_failed.1', 'M7_fail_opA05']	['Partial']	8	['X', 'X']	['X', 'X']	parttypeB_7
48	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	9	['X', 'X']	[InternalEvent('Failure_finished')]	parttypeB_7
49	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	10	['X', 'X']	['X', 'X']	parttypeA_6
50	['C01_M7_workComp_idleSt']	['M7_workComp', 'M7_opA05']	['Partial']	10	['X', 'X']	['X', 'X']	parttypeB_8

Figure 41: Portion of the simulation output table

In each second of the simulation, the file records the state each statechart enters, specifically in the columns *controller_state* for the controller, *machine_state* for the machine, and *buffer_state* for the buffer. The controller state will consistently display as idle because the statechart cycles through the entire queue at each time instant. If a statechart transitions through multiple states within the same second, only the final state is recorded in the main output. To mitigate this limitation for the machine, the output also logs all events generated by the statechart each second, the column *part_event* tracks events related to working parts, while *fail_event* records any failures. By capturing the sequence of events, the log enables precise tracking of the machine's behavior, even when it transitions through multiple states within a brief interval.

Additionally, the simulation tracks and logs the buffer level every second in the *parts_in_buffer* column. This data, drawn directly from the statechart's context, ensures precise real-time monitoring of buffer capacity. By recording the buffer level, the output allows for a detailed analysis of part movement through the system, checks if the buffer nears its capacity limits, and evaluates the effectiveness of release policies in preventing overloading.

Lastly, each second, the controller's selected part is logged in the *part_chosen* column of the Excel file. This detail is particularly relevant when the machine initiates new work, as it captures which part

the controller selected and subsequently processed. This precise tracking provides full traceability of the decision-making process, offering clear insight into the controller's actions and part prioritization.

Together, the Excel output provides a comprehensive view of the system's behaviour, capturing state transitions, event processing, buffer levels, and part selections in real time. This data is critical for the validation phase, as it serves as input for the subsequent stage, where the accuracy of the statechart system and simulation in replicating the real behaviour of the system in a controlled, verified environment will be assessed.

5. Validation

Following the development and formulation of a system governed by the controller, which implements conwip and dispatching policies for the selection and processing of parts, a comprehensive validation method was introduced using Siemens' Plant Simulation software. This software enabled the recreation of the original system architecture, comprising a part generator, buffer, processing machine, and final sink, while preserving the operational characteristics defined by the state chart model. The objective was to verify the consistency and correctness of the controller's implementation under realistic simulation conditions.

Through the Plant Simulation model, the system was accurately mirrored, ensuring that the same processing rules, buffer capacities, and throughput limits were applied. The output generated by Plant Simulation was subsequently used to provide input events to the Sismic simulation code. This allowed for a direct comparison between the sequence of events produced by Siemens' software and the sequence generated by the state chart process implementation, ensuring both systems operated within the same time window (simulation duration).

To facilitate this comparison, three distinct Python codes were developed:

- **System KPI Evaluation**: The first code was designed to calculate the key performance indicators (KPIs) of the system, focusing on metrics such as the time each element remains in a specific state (e.g., waiting, processing, or failure), the number of finished parts, the overall system throughput, and any characteristics observed in the production cycle.
- **Simulation Output Comparison**: The second code was responsible for comparing the output logs of both the Sismic simulation and the Plant Simulation model. By aligning the event sequences generated by both simulations, any discrepancies could be identified and analysed.
- **Buffer Level Monitoring**: The third one was developed to track buffer levels throughout the simulation on a time-step basis. By monitoring buffer occupancy every second when a new event occurred in the object, it was possible to assess the system's adherence to buffer constraints, ensuring that neither overflows nor underutilizations took place.

After the development of these codes, a series of experiments was conducted. These experiments aimed to stress-test the state chart simulation code by subjecting it to various input event scenarios, such as fluctuating part arrival rates, machine breakdowns, and changes in buffer capacities. The results demonstrated the validity and versatility of the model, confirming its ability to handle different input conditions reliably.

These experiments provided strong evidence of the robustness and adaptability of the implementation, highlighting its potential for broader applications across different systems settings and event configurations.

5.1. Siemens Plant Simulation system

Siemens Plant Simulation is an advanced 2D/3D simulation tool that enables the modelling and analysis of production systems and processes in a highly efficient and structured manner. Its object-oriented architecture allows users to build detailed, hierarchical models of both discrete and continuous manufacturing operations. With integrated 2D/3D visualization, users can simulate complex systems using external CAD data and large-scale models without compromising performance.

The software's architecture supports modularity through inheritance and hierarchy, making it easier to manage complex simulations. Additionally, its open system design allows for integration with various Siemens applications and interfaces, facilitating seamless data exchange. Plant Simulation also includes a range of analysis tools for performance evaluation, including bottleneck detection, resource utilization, and cost analysis, helping users optimize their systems through advanced simulation techniques [35].

5.1.1. System composition

As previously discussed, the system described in the previous chapter has been faithfully recreated in Plant Simulation using state chart models for each of its components. This recreation ensures consistency between the conceptual design and its simulation in a virtual environment. The system is composed of four main elements:

- Part generator
- Buffer
- Machine
- Final sink

In addition to replicating the system structure, several custom methods, designed SimTalk codes, have been implemented within the simulation to implement policies and generate a detailed final report. This report logs all events occurring in the system, listing them chronologically for each specific

element during the simulation runtime. This level of detail allows for a thorough analysis of the system's dynamic behaviour, ensuring a deeper understanding of its performance.

The complete system is depicted in the figure below:

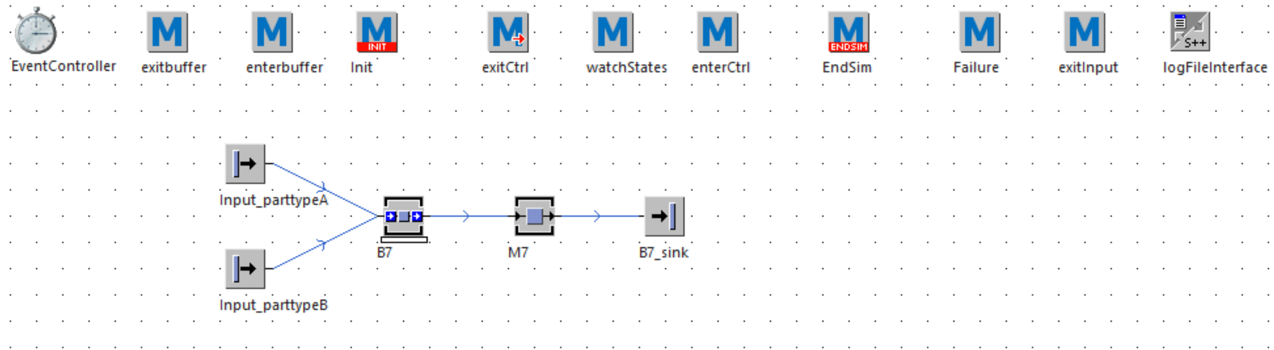

Figure 42: Plant Simulation Model layout

5.1.2. System elements

In the following section, each component and characteristics of the system will be analysed in detail, focusing on the specific characteristics that were configured to ensure consistency between the statecharts developed and the simulation performed in the software. The components are as follows:

i. MU parts

The system, as previously described, supports two types of parts: A and B. If a different type of part is detected, an error is signalled, and the decision-making cycle is restarted. For part generation, Siemens provides the capability to create a hierarchical structure, with "Part" as the parent entity and the specific types, A and B, as its child entities.

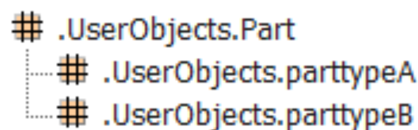

Figure 43: Part definition hierarchy

As illustrated in the adjacent figure, the parts generated and processed within the system have been defined in this manner.

Moreover, each part has different characteristics that are defined by user-defined attributes. An illustrative example of this phenomenon can be observed in the subsequent figure.

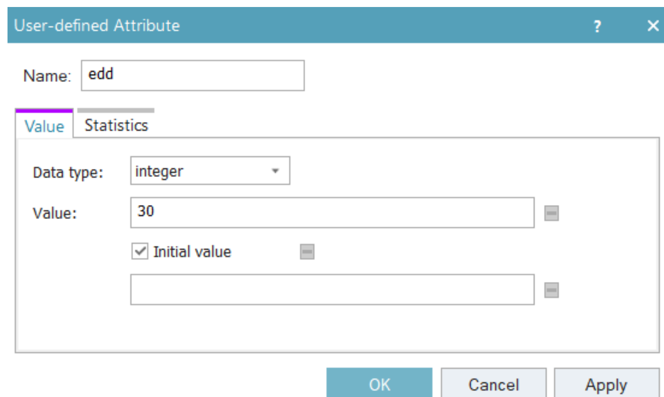

Figure 44: Part attribute

In this context, the attribute "*edd*" represents the time within which a specific part must be completed, and it has been established to implement the EDD policy.

ii. Parts generator

In the system, specific entity generators have been defined for each part type to facilitate reuse and modification. Each generator is responsible for introducing new parts into the system for processing, ensuring that the correct type is generated according to the established parameters. This modular design streamlines the part generation process and enhances the system's adaptability to changing production requirements.

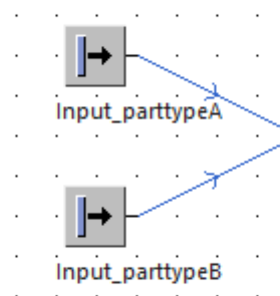

Figure 45: Input generation

The attributes selected during the various experiments are illustrated in the following figure:

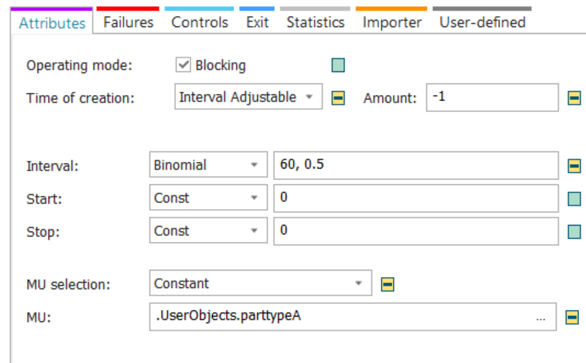

Figure 46: Part generator settings

The primary parameter is the "time of creation", selected as an "adjustable interval" type to facilitate the modification of different generation intervals. In this example, the generation of parts of type A is reported, specifically utilizing a binomial distribution for part generation.

Finally, it is important to note that the behaviour of this element is linked to that of the subsequent buffer. Parts are generated in accordance with the buffer's capacity availability. Specifically, if the buffer is in a *blocked* or *full* state, the generator enters an *idle* state. Furthermore, the generation logic differs based on the buffer's status. If, at a particular moment, a part is generated and processing in the machine begins, the part will be generated before processing if the buffer is empty; otherwise, it will be generated after processing has started if parts are already present in the buffer. This operational concept is crucial for accurately evaluating the parts to be processed according to the dispatching policies.

Moreover, in the simulation that implements the EDD policy, this component plays a critical role in managing the "*edd"* attribute of each part. Specifically, when a part is generated, the attribute is modified using the following calculation:

$$edd = edd_baseline + Uniform\ (x,y,z) + current\ simulation\ time$$

In this formula, "*edd_baseline*" represents a predetermined starting point. The uniform (x, y, z) component introduces randomness, allowing for variability in the earliest due dates and helping to validate the state chart model by simulating a range of scenarios. Finally, "current simulation time" ensures that the calculated EDD is always greater than the present moment, maintaining consistency within the model. This combination effectively generates realistic due dates for parts within the simulation.

iii. Buffer

The second element of the system is the buffer, whose primary function is to store generated parts while they await processing by the machine. Once the machine becomes available for processing and is in an idle state, the buffer releases one part, if available, to be processed.

Moreover, as previously mentioned, the buffer also influences the part generation process by the part generator, as it can block generation if it is in a *blocked* or *full* state.

The fundamental parameters for defining the behaviour of this element are two:

- Capacity: This refers to the maximum number of parts that the buffer can hold at any given time.
- Dispatching Policy: This defines the rules for determining the order in which parts are processed by the machine.

The two attributes are illustrated in the figure below:

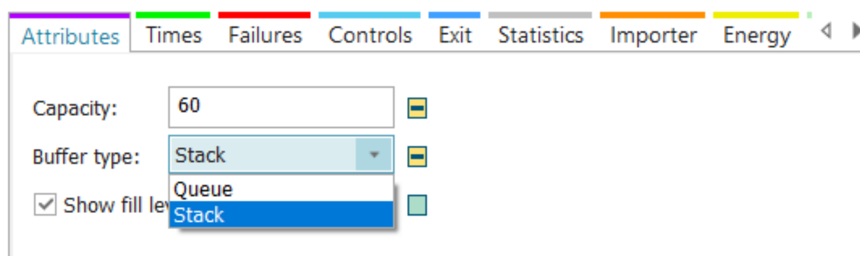

Figure 47: Buffer settings

In particular, the buffer type attribute allows for the implementation of two policies: FIFO (First-In, First-Out) and LIFO (Last-In, First-Out). The logic used in each case pertains to how the generated parts are inserted upon their arrival. The queue type adds parts to the end of the line, adhering to a FIFO logic, while the stack type allows new parts to be added to the beginning of the line, prioritizing them for processing in accordance with LIFO logic.

On the other hand, to validate the Earliest Due Date (EDD) policy, a code was implemented at the buffer's output. This method selects the part with the smallest *edd* attribute among those contained in the buffer, thereby effectively enforcing the policy.

```
for var i := 1 to t.yDim
    var part := t[1,i]
    var EddTime := part.edd

    if EddTime < minEdd
       minEdd := EddTime
        selectedPart := part
    end
```

Figure 48: Buffer exit method

Additionally, to monitor the buffer level, the updated quantity of parts is printed in the final output each time parts enter or exit the buffer. The attribute *"?.NumMU"* provides access to the current number of parts in the buffer at any given moment.

iv. Machine

Connected to the buffer and serving as the focal point of the system is the machine. It is responsible for processing the parts that enter the system. When in an idle state, the machine takes apart from the buffer according to the defined policy logic and processes it based on the time required for that specific part.

This is the most complex element of the system, as it can exist in multiple states that occur with random frequency:

- *Working*: The machine is actively processing parts.
- *Idle*: The machine is available but not currently processing any parts.
- *Failure*: The machine is non-operational due to a malfunction

To represent the occurrence of these states in Plant Simulation, several interchangeable parameters have been utilized.

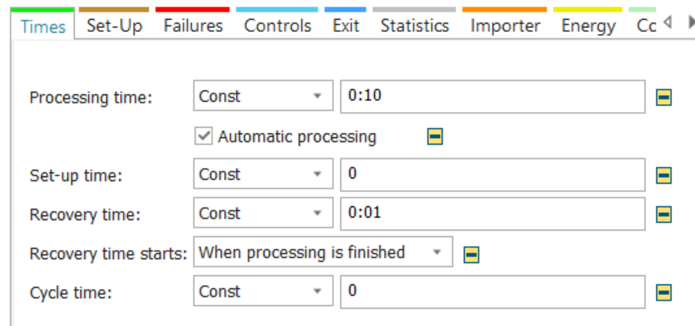

Figure 49: Machine settings

As shown in the figure, a constant “processing time” of 10 seconds has been assigned for each type of part. Additionally, it is important to note that after a part has been processed, there is a “recovery time” of 1 second (setup time), consistent with the statechart definitions within the system. This recovery time represents the idle period between one processing cycle and the next.

Another key factor in the formalization of the machine is the occurrence of failures. Siemens Plant Simulation provides an interface for creating and managing failures, as shown in the screen below:

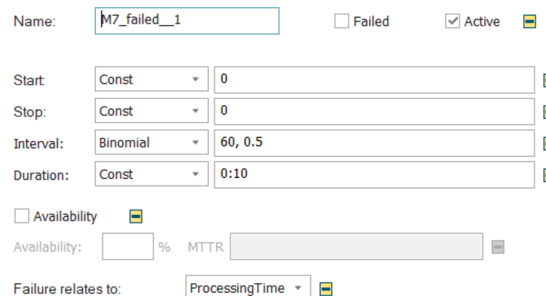

Figure 50: Machine failure settings

To define a failure type, the following parameters have been used:

- Occurrence distribution: In this example, a binomial distribution (60, 0.5) has been applied, meaning that a failure occurs after 60 processing cycles with a probability of 50% for each cycle.
- Duration: The time during which the machine remains in a failure state.
- Occurrence during processing time: Failures are set to occur during the active processing of parts, interrupting the machine's operation.

Additionally, as defined in the statecharts, whenever a failure occurs during processing, the part inside the machine, even if partially processed, is reworked from the beginning, meaning it undergoes the full processing time again. This action has been implemented using the following method:

```
(failureIsStarting: boolean; profileName: string)
is
do
	@.move(?)
end;
```

Figure 51: Machine failure method

The code is triggered each time a failure ends, and using the command "*@.move*", the part is reintroduced into the machine to restart processing from the beginning.

v. Final Sink

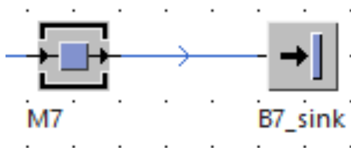

Figure 52: Final sink

The sink is the final element of the system, responsible for collecting the finished parts from the preceding machine. It also tracks the number of processed parts, which are used to calculate the final KPIs during the validation phase.

vi. Event Controller

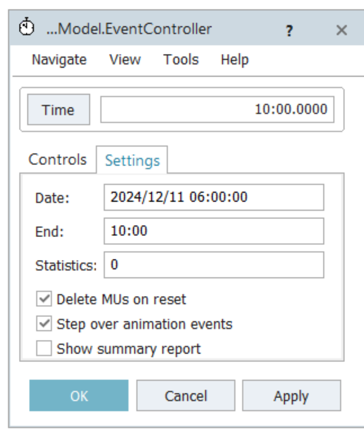

Figure 53: Event Controller settings

The event controller is the component that allows the user to set the simulation start date and duration. By managing the flow of events within the system, it ensures that the simulation adheres to the specified timeline. This functionality is useful for testing different operational scenarios, as it enables the user to define how long the system should run and at what intervals key events occur, thus allowing for precise control over the simulation's execution and results.

5.1.3. System summary

The system modelled in Siemens Plant Simulation replicates the logic and structure of the statechart-based design, focusing on four main components: the part generator, buffer, machine, and sink. Each element has been carefully defined to ensure consistency with the overall system.

The part generator creates two types of parts, A and B, with intervals that can be easily adjusted. Part generation is tied to the buffer's availability; if the buffer is full or blocked, the generator stops until space becomes available. The buffer stores parts awaiting processing and operates with a set capacity and dispatching policy, which can be either FIFO, LIFO or EDD, determining the order in which parts are processed.

The machine is responsible for processing parts and can alternate between working, idle, and failure states. Failures are modelled to occur randomly, based on a binomial distribution, and when a failure happens during processing, the part is reprocessed from the beginning after recovery. Lastly, the sink collects the finished parts and keeps track of the total processed, which is used for final performance assessments.

Each of these elements has been implemented with careful attention to detail, ensuring that the simulation mirrors the real-world system, allowing for accurate validation and performance measurement.

5.1.4. Final Output

5.1.4.1. Log File

To instantiate the Sismic simulation code with the events related to incoming parts and failures occurring during the simulation, the logfile.txt generated by Plant Simulation is utilized. This log file contains a sequence of events recorded throughout the production line simulation, where each line represents a specific event and includes essential information regarding the actions performed by the various components of the line. Each line can represent two types of events: an entry or exit from a factory object and an event related to the system’s state or one of its parts, specifically, failures that may occur in the factory objects during the simulation.

Below is depicted the first category of events, concerning the entry or exit from a factory object:

```
parttypeA_1, 0, place, exit, Input_parttypeA, 84
parttypeA_1, 0, place, enter, B7
parttypeA_1, 0, Buffer_level, enter, 1
parttypeA_1, 0, Buffer_List, enter, [*.UserObjects.parttypeA:1]
parttypeB_1, 0, place, exit, Input_parttypeB, 57
parttypeB_1, 0, place, enter, B7
parttypeB_1, 0, Buffer_level, enter, 2
parttypeB_1, 0, Buffer_List, enter,[*.UserObjects.parttypeA:1][*.UserObjects.parttypeB:1]
parttypeB_1, 0, place, enter, M7
parttypeB_1, 0, place, exit, B7
parttypeB_1, 0, Buffer_level, exit, 1
```

Figure 54: Log structure

The description of the elements presents in each line of the log file, concerning events of entry and exit from stations, is as follows:

- **Part or Component:** Identifies the type of part or component involved in the event.
- **Time:** Represents the moment at which the event occurred in seconds. • Mode: Describes the type of event.
- **Event:** Indicates the specific action or event undertaken by the part, it can be either “enter” or “exit” related to a machine or a buffer.
- **Location:** Specifies the place or destination associated with the event, as a specific machine or a buffer.

In the logs of the simulations where the EDD policy has been implemented, two new elements are present:

- **Edd time:** Indicates the due date of each part generated and introduced into the system.
- **Buffer_list:** This object encapsulates all the elements present in the buffer when an event occurs, and it is useful for monitoring the buffer in more complex policies.

Furthermore, it is important to note that for events concerning the buffer, the current fill level of the buffer is also recorded.

On the other hand, below is a line of the log file line regarding the failure state of a machine:

```
M7, 400, states, failure, true
M7, 410, states, failure, false
```

Figure 55: Log failure structure

- **Location:** Identifies the specific machine where the event occurred. • Time: Indicates the exact time of the event, measured in seconds.
- **Mode:** Specifies the nature of the event.
- **Event:** Describes the category or type of the event, in this instance, it pertains to a "failure."
- **Condition:** Indicates the state associated with the event. In this context, "true" signifies the initiation of the failure, while "false" denotes its conclusion.

Understanding the log file format and providing all necessary inputs are crucial for achieving alignment and consistency between the Sismic simulation code, which is based on the statecharts, and the Plant Simulation system. Therefore, it is essential for the user to possess a solid familiarity with the log file structure. This knowledge ensures proper interaction with the code, enables accurate data analysis, and is vital for conducting a thorough validation of the entire system's construction.

5.1.4.2. KPI Table

Siemens Plant Simulation provides a report that summarizes the data related to the results of the simulation. For example, the following table represents the percentage of time that various elements remain in a certain state relative to the total simulation time.

Object	Working	Set-up	Waiting	Blocked	Powering up/down	Failed	Stopped	Paused	Unplanned	Portion
Input_parttypeA	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	
M7	64.67%	0.00%	15.33%	0.00%	0.00%	20.00%	0.00%	0.00%	0.00%	
B7	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	
B7_sink	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	
Input_parttypeB	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	

Figure 56: Plant Simulation statistics report

Furthermore, there are additional useful metrics for the final validation, such as the total number of processed and finished parts in the machine. All this data has been collected to document the performance of each experiment in an Excel data frame, which will be used in the validation codes described in the following chapters.

KPI	EXP1_FIFO	EXP1_LIFO	EXP2_FIFO	EXP2_LIFO	EXP3_FIFO	EXP3_LIFO	EXP4_FIFO	EXP4_LIFO	EXP5_FIFO	EXP5_LIFO	EXP6_FIFO	EXP6_LIFO
B-Full	330	330	340	340	215	215	192	192	256	256	0	0
B-Partial	270	270	260	260	385	385	408	408	344	344	227	227
B-Empty	0	0	0	0	0	0	0	0	0	0	373	373
M-Working	381	381	348	348	409	409	432	432	432	432	388	388
M-Idle	29	29	22	22	31	31	38	38	38	38	92	92
M-Failure	190	190	230	230	160	160	130	130	130	130	120	120
M-Parts finished	29	29	22	22	31	31	38	38	38	38	34	34
M-Parts reworked	19	19	23	23	16	16	13	13	13	13	12	12

Figure 57: KPI table

In particular, the metrics calculated for each simulation experiment are as follows:

- **Buffer states**: % state * total simulation time
- **Machine states**: % state * total simulation time
- **Finished parts**: collected from the report
- **Reworked parts**: Processed parts - finished parts

5.2. Validation code

To compare the outputs generated from the results (Excel file) of the statechart interpretation that describes the system, and the log file produced by Plant Simulation, three codes have been developed to facilitate a detailed and consistent validation. These codes are:

1. **KPI Evaluation Code**: Assesses Key Performance Indicators (KPIs) such as time spent in each state and the number of completed parts.
2. **Output Comparison Code**: Compares outputs from the Sismic simulation and the Plant Simulation log file to identify discrepancies.
3. **Buffer Level Monitoring Code**: Tracks buffer levels at each time interval to ensure alignment with dispatching policies.

These codes collectively enable a comprehensive validation process, ensuring the accuracy and reliability of the simulation results.

5.2.1. KPI Evaluation Code

The KPI Evaluation Code processes data from the simulation log file and an Excel file containing expected KPIs to evaluate the performance of the system modelled in Plant Simulation. It calculates metrics such as the time spent in various states, the number of parts processed, and the incidence of failures.

Finally, it employs the *unittest* library to automate validation tests, comparing simulation results from statechart code against Plant simulation output. Overall, the code enables comprehensive performance analysis and validation of the simulation model.

Input:

The analysis takes several inputs to employ the validation:

1. Log File: Generated by Plant Simulation, this file records events related to the production line, including part entries, exits, and failures.
2. KPI table (Plant Simulation): This file contains Key Performance Indicators (KPIs) table.
3. *Experiment_simulation*: This dataframe contains all the output data generated by the Sismic code, which is based on the statechart simulations.

4. Simulation Duration Time: The code defines a specific timeframe for evaluating the system's performance, ensuring that all metrics are calculated within the same duration of the simulation.

Code:

In the first part, the code processes the output data generated by the Sismic simulation to derive Key Performance Indicators (KPIs) that are used for assessing the production system's efficiency and reliability. It begins by reading the final simulation results stored in an Excel file. This file contains a structured representation of various metrics, including the elements operational states and events.

By analyzing the transitions between states, the code calculates the total duration spent in each mode. The function used is called *calculate_state_time*, and it is represented in the following picture:

```
def calculate_state_time(df, state_column):
```

Figure 58: Calculate state time function

It takes as input the dataframe and the corresponding column for the system element to be analyzed, allowing for a detailed analysis of its performance.

The result obtained from applying the function is used to calculate all relevant metrics, as illustrated in the following figure.

```
working_time_M7 = sum(duration for state, duration in machine_total.items() if 'M7_workComp' in state)
idle_time_M7 = sum(duration for state, duration in machine_total.items() if 'M7_idleComp' in state)
failed_time_M7 = sum(duration for state, duration in machine_total.items() if 'M7_failed.1' in state)
```

Figure 59: Machine state time calculation from Sismic output

In addition to time states, other performance metrics were calculated, such as the number of processed, finished, and reworked parts after failures. The results were divided by part type to validate the final outcome with greater accuracy. In this case, a function called *count_unique_events* were used, which, by taking as input the dataframe containing the list of events occurring in a specific element every second, returns the number of occurrences of these events.

```
finished_parts_M7_A = working_parts_M7_A
finished_parts_M7_B = working_parts_M7_B
finished_parts = finished_parts_M7_A + finished_parts_M7_B
```

Figure 60: Finished parts calculation

As shown in the figure, the finished parts are determined by the number of occurrences of the "exit" part event from the machine for each specific part type. Once calculated, they are summed to determine the total number of finished parts.

On the other hand, to determine the reworked parts after failures for each specific part type, a function was used to count the number of occurrences of entry into a state. This is because failure events do not discriminate between whether a part of type A or type B is present in the machine. In fact, the reworked parts are calculated as the total number of failure occurrences minus the cases where a failure started but did not finish.

```
reworked_parts_M7_A = failed_count_M7_A - count_part_f_A
reworked_parts_M7_B = failed_count_M7_B - count_part_f_B
reworked_parts_M7 = reworked_parts_M7_A + reworked_parts_M7_B
```

Figure 61: Reworked parts calculation

In the figure, the formulas are shown along with the total number of reworked parts, which corresponds to the sum of reworked parts for each specific part type.

After gathering these insights from the Sismic simulation output, the code turns its attention to the log file generated by Plant Simulation. This log file is rich with data regarding real-time events in the production line, detailing part entries, exits, and any failures that occurred during the simulation.

As previously mentioned, the output of the simulation performed using Siemens Plant Simulation is saved in two formats. In the first, a dataframe is created from the report results, summarizing all calculations for each experiment.

However, not all data is available from the report, particularly those specific to the machine regarding the type of part processed and reworked. For this reason, a script was developed to read the entire log file and, by interpreting the events, return two key metrics: the number of finished and processed parts for each part type, as well as the processing time dedicated to each specific part type and the corresponding types of failures.

The methodology used to calculate these metrics involves using a count function to track all "enter" events in the machine while subtracting occurrences of failures to avoid counting reworked parts. The reworked parts are determined by the number of failures for each specific part type.

The following formulas were used to calculate the processing times:

```
work_time_A = np.sum(np.abs(exit_A - enter_A))
work_time_B = np.sum(np.abs(exit_B - enter_B))
failure_A = np.sum(np.abs(failure_F_A - failure_T_A))
failure_B = np.sum(np.abs(failure_F_B - failure_T_B))
```

Figure 62: Machine state time calculation from Plant simulation log file

The start and end times of a specific state were stored in separate arrays for each state. These times were then used to calculate the total duration of the state by summing the differences.

Output

The final step involves a thorough comparison between the KPIs derived from the Sismic simulation and those extracted from the Plant Simulation log file. This validation process is crucial, as it ensures that the simulation results are accurate and reflective of the system's actual performance. The validated KPIs include:

- **Buffer KPIs:**
    - B-Full: Total duration the buffer was full.
    - B-Partial: Total duration the buffer was partially filled.
    - B-Empty: Total duration the buffer was empty.
- **Machine KPIs:**
    - M-Working: Total time the machine was actively working.
    - M-Idle: Total time the machine was idle.
    - M-Failure: Total time the machine experienced failures.
- **Processed Parts:**
    - M-Parts Finished (A): Total number of part type A processed successfully.
    - M-Parts Finished (B): Total number of part type B processed successfully.
    - M-Parts Reworked (A): Total number of part type A that required rework due to failures.

- M-Parts Reworked (B): Total number of part type B that required rework due to failures.

To facilitate the comparison of Key Performance Indicators (KPIs) and automate the validation process, the *Unittest* library has been employed. This library allows for the systematic testing of the calculated outputs against predefined benchmarks from the Plant Simulation log file. By creating a dedicated test class, various assertions are made to compare the results of the Sismic simulation with expected values for each KPI.

The possible outcomes of the Validation KPI code from *Unittest* are:

- Success (OK): The test has passed all assertions without errors. This is the desired outcome, indicating that the behaviour of the code under test conforms to expectations.

```
----------------------------------------------------
----------------------
Ran 14 tests in 0.187s

OK
```

Figure 63: Unittest function output

- Failure (FAIL): The test has resulted in failure, indicating that at least one of the assertions did not successfully verify. This may indicate a bug or a discrepancy between the expected and actual behaviour of the code.
- Error (ERROR): An error occurred during the test execution, but not necessarily due to a violation of assertions. For example, there might be an unhandled exception in the test code itself or in the code under test.

Not all KPIs used the *assertEqual* function to compare the metrics. In some cases, as shown in the following figure, the *almostEqual* method was used with a unit delta of 1 to account for some decimal rounding in the calculation of the KPIs from Plant Simulation in the KPI table.

```
def test_B_full(self):
    self.assertAlmostEqual(Full_B7, self.validation_experiment.loc['B-Full'], delta = 1.0)
```

Figure 64: Buffer validation function

In this example, the buffer's dwell time in a full state can be considered equal with a variance of 1 unit of time (one second).

5.2.2. Validation events order

This code compares the results from the state chart model simulation and a log file generated by Plant Simulation for a machine ("M7"). The goal is to validate whether events such as part entries, exits, and machine failures match between the simulated and actual data. It processes and synchronizes the data from both sources, filters relevant events, and checks for matches between the two datasets. Key metrics such as part movements and machine failure states are tracked, and the results are stored in an Excel file for making the analysis.

Input:

The code takes two key inputs: an Excel file containing the simulation results generated in Sismic code and a log file produced by Plant Simulation.

Code:

The process carried out by the code is aimed at thoroughly comparing the performance of the Sismic-based simulation and the actual results logged from the Plant Simulation for the machine. Initially, the log file from Plant Simulation is converted into a structured DataFrame, filtered to focus solely on machine-related events such as part entries, exits, and failures. These log events are crucial because they offer insight into how the machine operated in the real-world scenario.

Once the log data is organized, both the log and simulation data undergo a rounding process to ensure synchronization of their respective timestamps. This is vital since slight differences in the time format can otherwise make direct comparisons difficult. After this synchronization, the log and simulation data are merged, allowing the code to match and compare the system states at each specific point in time:

```
df_merged = pd.merge(df_log_m7, df_state, on='Rounded_time', how='inner')
```

Figure 65: File merging for comparison

Then, the code then focuses on cleaning the data by dropping redundant rows and columns. In particular, the "exit" and "enter" events that occur after a failure, which coincide with the re-entry of the part to be reworked in the machine, were removed. The following commands are used:

```python
false_indices = df_check[df_check['5'].str.strip() == 'false'].index
rows_to_drop = false_indices + 1
rows_to_drop = rows_to_drop.union(false_indices + 2)
df_check = df_check.drop(rows_to_drop, errors='ignore')
```

Figure 66: Event removal after failure

These events should not be included in the final comparison because, in the statechart, the part is not re-entered into the machine with an "enter" event. Instead, the part remains in the machine during the failure and resumes processing once the failure ends.

In the end, the dataframe used for the final analysis has the following structure:

1	2	4	5	Machine_state	Part_event	Fail_event	Part_chosen
parttypeB_1	0	enter	M7	['M7_workComp', 'M7_opB04']	[InternalEvent('dequeue'), InternalEvent('Started_opB04')]	['X', 'X']	parttypeB_1
parttypeB_1	10	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opB04')]	['X', 'X']	parttypeA_1
parttypeA_1	11	enter	M7	['M7_workComp', 'M7_opA05']	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_1
parttypeA_1	21	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opA05')]	['X', 'X']	['X', 'X']
parttypeA_2	33	enter	M7	['M7_workComp', 'M7_opA05']	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_2
parttypeA_2	43	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opA05')]	['X', 'X']	['X', 'X']
parttypeB_2	45	enter	M7	['M7_workComp', 'M7_opB04']	[InternalEvent('dequeue'), InternalEvent('Started_opB04')]	['X', 'X']	parttypeB_2
M7	50	failure	true	['M7_failed.1', 'M7_fail_opB04']	['X', 'X']	[InternalEvent('Failure_started')]	['X', 'X']
M7	60	failure	false	['M7_workComp', 'M7_opB04']	['X', 'X']	[InternalEvent('Failure_finished')]	parttypeA_3
parttypeB_2	70	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opB04')]	['X', 'X']	parttypeA_3
parttypeA_3	71	enter	M7	['M7_workComp', 'M7_opA05']	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_3
parttypeA_3	81	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opA05')]	['X', 'X']	['X', 'X']
parttypeB_3	85	enter	M7	['M7_workComp', 'M7_opB04']	[InternalEvent('dequeue'), InternalEvent('Started_opB04')]	['X', 'X']	parttypeB_3
parttypeB_3	95	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opB04')]	['X', 'X']	parttypeA_4
parttypeA_4	96	enter	M7	['M7_workComp', 'M7_opA05']	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_4
M7	100	failure	true	['M7_failed.1', 'M7_fail_opA05']	['X', 'X']	[InternalEvent('Failure_started')]	['X', 'X']
M7	110	failure	false	['M7_workComp', 'M7_opA05']	['X', 'X']	[InternalEvent('Failure_finished')]	['X', 'X']
parttypeA_4	120	exit	M7	['M7_idleComp']	[InternalEvent('Finished_opA05')]	['X', 'X']	parttypeA_5
parttypeA_5	121	enter	M7	['M7_workComp', 'M7_opA05']	[InternalEvent('dequeue'), InternalEvent('Started_opA05')]	['X', 'X']	parttypeA_5

Figure 67: File for comparison

As we can see in the figure, the first columns with numerical names represent the events in the log file of Plant Simulation, while the subsequent columns store the states and events that occur in the machine, as derived from the simulation using statecharts.

Once the data frame was prepared with the necessary information for the final analysis, two comparisons were constructed to validate the alignment of the simulation results from the two software packages. These comparisons are:

1. **State Comparisons**: This comparison focuses on the machine states at various points in time, ensuring that the machine's operational states in the Python simulation align with those recorded in the Plant Simulation log file. Key aspects include:
    - *State Tracking:* It tracks the various states of the machine throughout the simulation time. The analysis verifies that the machine state accurately reflects these transitions, ensuring that the simulation effectively captures the dynamics of the production process.

- *Failure Identification:* The code verifies that occurrences of failures in the simulation correspond with those logged. By analyzing the states before and after failure events, we ensure that the simulation accurately represents the machine's operational interruptions and recovery processes.

2. **Event Comparisons**: This comparison evaluates the specific events logged during the simulation, assessing the consistency between the two data sources. Important highlights include:
   - *Part Flow Events:* The analysis examines events related to the entering and exiting of parts, comparing the counts and timing in both simulations. This helps confirm that the flow of materials through the machine is consistent across both platforms, which is crucial for understanding throughput and operational efficiency.
   - *Failure Event Tracking:* The code checks for both the start and end of failure events, ensuring that the logged failures in Plant Simulation align with those in the Python simulation.

By conducting these comprehensive comparisons, we can validate the simulation results and ensure they accurately reflect the system's actual performance. This validation process not only enhances the accuracy of the Sismic simulation and the correctness of the statechart construction that comprises the system but also provides insights into areas for improvement, ensuring that the production system operates effectively and efficiently.

Output

The results generated by the code provide valuable insights into the performance of the simulation through the two comparisons conducted. Each print statement summarizing the outputs is crucial for understanding the alignment between the simulation results and the expected operational events.

1. **State Check Results**: The output for this comparison indicates the number of matches found for part entry and exit events. Specifically, the print statements show:
   - *Total number of matched rows:* This figure represents how many times the state transitions logged in the simulation correspond with those recorded in the Plant Simulation log. If the possible matches coincide with those counted, it indicates a complete correspondence between the two systems.
   - *Rows matching specific conditions:* The detailed counts for parts entering the system (both parttypeA and parttypeB), exiting the system, and failure events provide further granularity.

```
Results State entrance check:

Maximum number of matches possible: 95
Total number of matched rows: 95
Rows matching 'enter' and 'parttypeA': 15
Rows matching 'enter' and 'parttypeB': 17
Rows matching 'exit' and 'idle': 31
Rows matching 'true' failure: 16
Rows matching 'false' failure: 16
```

Figure 68: State check results output

The figure shows the terminal output that the user can view to check the matching between the states and their distribution.

2. **Event Check Results**: This output focuses on specific operational events within the system:
   - *Matching parts:* This metric shows how many parts processed in the simulation match the expected outcomes based on the operational data. It allows for verifying that the dispatching policy implemented by the simulation code, based on the statecharts, is correct and effectively applied in the selection of parts to be processed.
   - *Counts for specific events:* The print statements for enter and exit events, as well as failure occurrences, help to diagnose the reliability of the machine operations. Discrepancies in these counts, especially for critical events like failure starts and finishes, may signal underlying problems in the state chart code or formulation that need further exploration.

```
Results events check:

Matching parts: 32
Total Worked parts: 32
Matching enter M7 events: 32
Matching exit M7 events: 31
Matching fail start M7 events: 16
Matching fail finish M7 events: 16
```

Figure 69: Events check results output

By analyzing these outputs, it is possible to validate the correct construction of the state charts for the machine and buffer controller, as well as the proper implementation of the SISMISC library and the dispatching policies in the selection of parts to be processed. This also ensures cohesion with the Plant Simulation software in terms of event timing and sequencing.

5.2.3. Validation buffer level

This code segment is designed to validate the buffer levels recorded in a production system simulation by comparing the results obtained from a Python simulation with the log data generated by Plant Simulation. It systematically processes and analyzes the data to confirm that the buffer levels documented at each time point are consistent across both datasets.

Input:

The code takes two key inputs: an Excel file containing the simulation results generated in state chart model code and a log file produced by Plant Simulation.

Code:

The code serves in validating the buffer levels within a production system. Once the input files are set, the code utilizes a function to read the log file and convert it into a Data Frame. After this initial processing, the code filters the log data to focus exclusively on buffer events, which are essential for understanding how many parts are present.

The next step involves cleaning the data by removing any duplicate entries from the log, retaining only the most recent state for each timestamp. This ensures that the buffer levels reflect the latest updates, thereby enhancing the reliability of the validation process. With the data prepared, the code merges the filtered log Data Frame with the simulation results based on the rounded timestamps, resulting in a comprehensive dataset ready for comparison.

Once the merging is complete, the code further refines the data by eliminating any unused columns, streamlining the dataset for the analysis. It then embarks on the validation of buffer levels by iterating through each row of the cleaned Data Frame.

```
for index, row in df_check.iterrows():
    if int(row['5'])== int(row['Parts_in_buffer']):
        count+=1
    else:
        print('\n Row not matched:\n', row, '\n')
```

Figure 70: Buffer level checking cycle

For each row, it compares the buffer levels recorded in the log file with those derived from the state chart simulation. If the levels match, a count is incremented; if not, the code prints a message indicating a mismatch.

Output

The code outputs the total number of rows compared along with the number of successful matches, providing a concise summary of the validation results.

```
Maximum number of matches possible: 94
Total number of matched rows: 94
```

Figure 71: Buffer kevel check results

In the figure above, the final terminal output is shown, indicating the possible rows that can match the buffer level, compared to those that match between the two simulations. If they coincide, we have complete alignment between the system defined by the state charts and Plant Simulation.

5.3. Experiments

Several experiments were conducted to validate the model defined with the state chart. These tests aimed to ensure that the model accurately represents the system's intended behavior, confirming its reliability across different scenarios and conditions.

5.3.1. Experiments setting

Several experiments were conducted by varying key parameters in Plant Simulation, specifically:

- the generation rates of type A and B parts,
- the frequency of machine failures.

These parameters, which directly affect production flow and system operations, serve, as outlined in the previous chapter, as inputs not only for Plant Simulation but also for the *Sismic* simulation code, 10 experiments were conducted: 6 to validate both FIFO and LIFO dispatching policies, and 4 focusing on special cases to validate the earliest due date policy. Each experiment has a simulation time of 600 (10 minutes). A summary of these experiments is provided in the following table:

Experiment Number	Input Part A	Input Part B	Machine Failure
1	10 seconds	6 seconds	Binomial 40, 0.5
2	10 seconds	6 seconds	Binomial 30, 0.5
3	8 seconds	12 seconds	Binomial 50, 0.5
4	8 seconds	12 seconds	Binomial 60, 0.5
5	Binomial 60, 0.5	Binomial 60, 0.5	Binomial 60, 0.5
6	Binomial 60, 0.5	Binomial 90, 0.5	Binomial 60, 0.5
7	Binomial 15, 0.5	Binomial 40, 0.5	Binomial 40, 0.5
8	Binomial 50, 0.5	Binomial 40, 0.5	Binomial 40, 0.5
9	Binomial 50, 0.5	Binomial 60, 0.5	Binomial 40, 0.5
10	Binomial 40, 0.5	Binomial 30, 0.5	Binomial 50, 0.5

Table 1: Input objects parameters

As is showed, processing times and event occurrences were modelled using integer duration or event-based binomial distributions. The binomial distribution generates discrete time intervals based on a probability model, ensuring that event timings are represented as whole seconds. This was necessary

to align with the *Sismic* code, which updates and operates with a frequency of one second, ensuring consistency between Plant Simulation and the statechart based system.

Additionally, to validate the Earliest Due Date (EDD) policy, two other parameters described in previous chapters were varied: the “due date baseline” and the “random factor” in the due date.

Experiment	Edd A Baseline	Edd B Baseline	Random Generation
7	40 seconds	15 seconds	uniform (1,0,50)
8	30 seconds	15 seconds	uniform (1,0,50)
9	10 seconds	30 seconds	uniform (1,0,30)
10	0	0	uniform (1,0,70)

Table 2: Input EDD parameters

As illustrated in the chart, the EDD baseline is defined as a constant time during part generation, while the random attribute is defined alongside the z-uniform parameter, introducing a random factor that affects the due date and thus renders the final completion date variable. This variability in due dates is essential for assessing the system’s responsiveness to diverse completion requirements across different products.

5.3.2. Experiments results

Throughout the experiments, a variety of system configurations were investigated, incorporating alterations in part generation rates and machine failure frequencies. This diversity in configurations permitted a comprehensive assessment of the system's responsiveness to disparate input conditions. Notably, all tests were successfully completed, thereby underscoring the resilience and the capabilities of the state chart system.

To validate the model, multiple procedures were employed. KPI comparison assessed key performance metrics, such as throughput and machine utilization, against established benchmarks, thereby confirming the model's adaptability to different operational scenarios. Event sequence alignment ensured that the timing and order of events aligned with real-world expectations, which is essential for effectively managing transitions during fluctuations in part generation and unexpected machine failures. Additionally, buffer level analysis continuously monitored occupancy throughout the simulation, ensuring that the system maintained optimal buffer levels even during variations in production.

The positive outcomes from these validation procedures underscored the flexibility and robustness of the state chart-based system. Its capability to implement various dispatching policies further emphasizes its adaptability, as it can accommodate a wide range of inputs and configurations while consistently maintaining strong performance metrics. This adaptability highlights the system's effectiveness in representing complex manufacturing dynamics. Overall, these findings affirm that the state chart model is a versatile tool for analysing and optimizing production processes. It demonstrates a proven ability to respond effectively to changing operational conditions, ensuring reliability and accuracy in implementing dispatching policies.

6. Conclusion

The objective of this thesis was to improve an existing simulation model for production systems by incorporating dispatching policies into a statechart-based framework. This research aimed to expand the original model, which previously concentrated solely on release policies, to include dispatching mechanisms as well. The implementation of effective dispatching policies is of paramount importance for the real-time prioritisation and the efficient flow of parts through machines and buffers, thereby ensuring seamless operations in complex manufacturing environments.

The key goal of this work was to develop a modular, flexible, and reusable statechart model that could be adapted to different production contexts while allowing for the application of various control policies. The statecharts developed in this thesis were specifically designed to facilitate the implementation of both release and dispatching policies. This approach provides a significant advantage in handling operational disruptions such as fluctuating part generation rates, machine breakdowns, and buffer shortages.

The main contributions of this thesis can be summarized as follows:

- **Modular Statechart Design**:

A primary objective of this project was the development of statecharts with a focus on modularity. By designing the statecharts for the controller, machine, and buffer to be adaptable and reusable across different production environments, this thesis offers a flexible framework that can be applied without needing significant changes to the underlying models. The modularity of the statecharts also allows for the easy application of different release and dispatching policies, providing users with the flexibility to test various operational strategies without modifying the core statechart architecture.

- **Integration of Dispatching Policies**:

This work demonstrated how dispatching policies can be integrated into a statechart-based simulation model. The dispatching policies considered in this thesis include FIFO, LIFO, and EDD, each of which provides different prioritization rules for part selection and machine processing. These policies were implemented alongside a CONWIP release policy to control the overall flow of parts into the system. The integration of both release and dispatching policies within the same framework allows for a more holistic approach to managing production processes, ensuring that parts are processed in the most efficient order while maintaining optimal system throughput.

- **Real-Time Decision-Making**:

A major strength of the statechart-based model developed in this thesis is its ability to make real-time decisions based on the current state of the system. The controller statechart plays a pivotal role in this regard, as it is responsible for selecting and dispatching parts to machines based on the real-time conditions of the production system. This ensures that the system can dynamically adapt to changes, such as unexpected machine failures or sudden fluctuations in part generation rates, without causing significant disruptions to the production process.

- **Validation through Simulation**:

To validate the effectiveness of the statechart model, the Python-based simulation was used to simulate various operational scenarios. These simulations tested the system under different conditions, such as varying part flow rates, machine states, and buffer capacities. The results were then compared with those obtained from Siemens Plant Simulation, a widely used tool in industrial production environments. The consistency of the results between the two simulation tools confirmed the accuracy and reliability of the statechart-based model. The experiments demonstrated that the model could effectively handle complex production dynamics, ensuring that both release and dispatching policies could be applied flexibly and efficiently.

- **Handling Disruptions**:

A key feature of the model is the statechart's ability to handle unexpected disruptions. Firstly, the controller statechart integrates error-handling mechanisms that prevent invalid operations from blocking the system. Additionally, the machine statechart is designed to manage breakdowns, ensuring that part information is preserved throughout the recovery process. This robustness enables the system to operate efficiently, even in the face of unforeseen events.

Looking ahead, several avenues for future research and development emerge from the findings of this thesis. One significant area of improvement is the application of more complex and diverse release and dispatching policies. While this thesis incorporated basic policies such as FIFO, LIFO, and EDD, future research could explore more advanced policies that are tailored to specific production requirements or scenarios. These could include hybrid policies that combine the strengths of multiple strategies, or policies designed to optimize for different objectives, such as reducing lead times, minimizing costs, or balancing resource utilization.

Additionally, expanding the statechart framework to model more intricate production systems represents another promising direction for further research. The current thesis focused on a system composed of a single machine, buffer, and controller; however, future work could apply these policies to a more complex system involving multiple machines, buffers, and part types. By doing so, the framework could better represent the diverse, interconnected nature of real-world manufacturing environments, offering even greater flexibility and control over production processes. This approach would allow for more realistic simulations of large-scale production systems, providing insights into how different release and dispatching policies perform in complex, multi-component environments.

In conclusion, this thesis has demonstrated the effectiveness of UML statecharts in modeling and simulating advanced production systems. The integration of dispatching policies into the statechart framework has introduced a new level of flexibility and control, ensuring that tasks are prioritized, and resources are allocated efficiently. The validation of the model through simulation confirmed its robustness and adaptability, making it a valuable tool for both academic research and industrial applications. Future enhancements, including the application of more sophisticated control policies and their deployment in more complex production systems, offer exciting possibilities for further optimization of manufacturing processes.

7. Bibliography

[1] "AMCI : Advanced Micro Controls Inc :: What is a PLC?" Accessed: Nov. 03, 2024. [Online]. Available: https://www.amci.com/industrial-automation-resources/plc-automation-tutorials/what-plc/

[2] M. G. Hudedmani, R. M. Umayal, S. K. Kabberalli, and R. Hittalamani, "Programmable Logic Controller (PLC) in Automation," *Advanced Journal of Graduate Research*, vol. 2, no. 1, pp. 37–45, May 2017, doi: 10.21467/ajgr.2.1.37-45.

[3] B. Vogel-Heuser, S. Braun, B. Kormann, and D. Friedrich, "Implementation and evaluation of UML as modeling notation in object oriented software engineering for machine and plant automation," in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, IFAC Secretariat, 2011, pp. 9151–9157. doi: 10.3182/20110828-6-IT-1002.01343.

[4] G. Łabiak, M. Adamski, J. Tkacz, M. Doligalski, and A. Bukowiec, "Role of UML modelling in discrete controller design," in *Proceedings - ICSEng 2011: International Conference on Systems Engineering*, 2011, pp. 480–481. doi: 10.1109/ICSEng.2011.97.

[5] R. S. Moura and L. A. Guedes, "Using basic Statechart to program industrial controllers," *Comput Stand Interfaces*, vol. 34, no. 1, pp. 60–67, Jan. 2012, doi: 10.1016/j.csi.2011.05.006.

[6] S. Si, A. Sinan, and S. Alhir, "Understanding the Unified Modeling Language (UML)." [Online]. Available: http://home.earthlink.net/~salhir

[7] C. Ansorge and I. Skender, "Introduction to Unified Modeling Language (UML)." Accessed: Nov. 03, 2024. [Online]. Available: https://www.gfa-group.de/web-archive/inspire/www.inspiration-westernbalkans.eu/5/9/5/3/7/7/Introduction_to_the_Unified_Modeling_Language__UML_.pdf

[8] G. P. Favini, "Introduzione a UML." Accessed: Nov. 03, 2024. [Online]. Available: http://www.cs.unibo.it/gabbri/MaterialeCorsi/1.introUML.favini.pdf

[9] Omg, "OMG ® Unified Modeling Language ® Standard - Version 2.5.1," 2009. [Online]. Available: https://www.omg.org/spec/UML/20161101/PrimitiveTypes.xmi

[10] F. Cicirelli, A. Furfaro, and L. Nigro, "Modelling and simulation of complex manufacturing systems using statechart-based actors," *Simul Model Pract Theory*, vol. 19, no. 2, pp. 685–703, Feb. 2011, doi: 10.1016/j.simpat.2010.10.010.

[11] M. Naughtono, J. Mcgrath, D. Heffernan, and M. Naughoto, "Real-time Software Modelling using Statecharts and Timed Automata Approaches," 2006.

[12] "UML state machine - Wikipedia." Accessed: Nov. 03, 2024. [Online]. Available: https://en.wikipedia.org/wiki/UML_state_machine

[13] A. Knapp and S. Merz, "Model Checking and Code Generation for UML State Machines and Collaborations." [Online]. Available: http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

[14] B. Vogel-Heuser, D. Friedrich, and D. Witsch, "Usability and benefits of UML for plant automation-some research results." [Online]. Available: https://www.researchgate.net/publication/242728690

[15] K. Lano and A. Evans, "Rigorous development in UML," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 1999, pp. 129–144. doi: 10.1007/978-3-540-49020-3_9.

[16] M. Stevenson, L. C. Hendry, and B. G. Kingsman, "A review of production planning and control: The applicability of key concepts to the make-to-order industry," Mar. 01, 2005. doi: 10.1080/0020754042000298520.

[17] J. W. M. Bertrand and J. C. Fransoo, "Operations management research methodologies using quantitative modeling," *International Journal of Operations and Production Management*, vol. 22, no. 2, pp. 241–264, 2002, doi: 10.1108/01443570210414338.

[18] F. Robert Jacobs, "Manufacturing Planning and Control for Supply Chain Management."

[19] W. J. Hopp and M. L. Spearman, *Factory Physics*, Third Edition. Waveland Press, Inc, 2008.

[20] S. Nigel, C. Stuart, and J. Robert, *Operations Management*, Sixth. 2010.

[21] D. Mourtzis, E. Vlachou, and N. Milas, "Industrial Big Data as a Result of IoT Adoption in Manufacturing," in *Procedia CIRP*, Elsevier B.V., 2016, pp. 290–295. doi: 10.1016/j.procir.2016.07.038.

[22] C. T. Maravelias and C. Sung, "Integration of production planning and scheduling: Overview, challenges and opportunities," *Comput Chem Eng*, vol. 33, no. 12, pp. 1919–1930, Dec. 2009, doi: 10.1016/j.compchemeng.2009.06.007.

[23] A. Bueno, M. Godinho Filho, and A. G. Frank, "Smart production planning and control in the Industry 4.0 context: A systematic literature review," *Comput Ind Eng*, vol. 149, Nov. 2020, doi: 10.1016/j.cie.2020.106774.

[24] S. B. Gershwin, "Design and operation of manufacturing systems: the control-point policy."

[25] J. H. BLACKSTONE, D. T. PHILLIPS, and G. L. HOGG, "A state-of-the-art survey of dispatching rules for manufacturing job shop operations," *Int J Prod Res*, vol. 20, no. 1, pp. 27–45, Jan. 1982, doi: 10.1080/00207548208947745.

[26] M. L. Pinedo, *Scheduling - Theory, Algorithms, and Systems*, Sixth Edition. Springer, 2022.

[27] S. Wang, J. Wan, D. Li, and C. Zhang, "Implementing Smart Factory of Industrie 4.0: An Outlook," *Int J Distrib Sens Netw*, vol. 2016, 2016, doi: 10.1155/2016/3159805.

[28] M. Fera, F. Fruggiero, A. Lambiase, G. Martino, and M. Elena, "Production Scheduling Approaches for Operations Management," in *Operations Management*, InTech, 2013. doi: 10.5772/55431.

[29] M. Urgo and W. Terkaj, "Formal Modelling of Release Control Policies as a plug-in for Performance Evaluation of Manufacturing Systems." [Online]. Available: http://ifcowl.openbimstandards.org/IFC4_ADD1#

[30] K. V. Morris Wright, T. S. Hoang, C. Snook, and M. Butler, "Formal Language Semantics for Triggered Enable Statecharts with a Run-to-Completion Scheduling," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Science and Business Media Deutschland GmbH, 2023, pp. 178–195. doi: 10.1007/978-3-031-47963-2_12.

[31] T. Tsai and R. Sato, "A UML model of agile production planning and control system," *Comput Ind*, vol. 53, no. 2, pp. 133–152, Feb. 2004, doi: 10.1016/j.compind.2003.07.003.

[32] "Virtual Learning Factory Toolkit | Virtual Learning Factory Toolkit." Accessed: Oct. 27, 2024. [Online]. Available: https://virtualfactory.gitbook.io/vlft

[33] A. Decan and T. Mens, "Sismic—A Python library for statechart execution and testing," *SoftwareX*, vol. 12, p. 100590, Jul. 2020, doi: 10.1016/J.SOFTX.2020.100590.

[34] "Sismic user manual — Sismic 1.6.8 documentation." Accessed: Oct. 20, 2024. [Online]. Available: https://sismic.readthedocs.io/en/latest/index.html

[35] "Plant simulation software | Siemens Software." Accessed: Oct. 27, 2024. [Online]. Available: https://plm.sw.siemens.com/en-US/tecnomatix/products/plant-simulation-software/

[36] "What is SCADA? Supervisory Control and Data Acquisition." Accessed: Nov. 05, 2024. [Online]. Available: https://inductiveautomation.com/resources/article/what-is-scada

[37] "What is a Distributed Control System (DCS) - ABB Distributed Control Systems - DCS." Accessed: Nov. 05, 2024. [Online]. Available: https://new.abb.com/control-systems/control-systems/what-is-a-distributed-control-system

[38] "IEC 61131-3 Protocol Overview - Real Time Automation, Inc." Accessed: Nov. 05, 2024. [Online]. Available: https://www.rtautomation.com/technologies/control-iec-61131-3/?srsltid=AfmBOorO-_K72pLmTe7-RKEdwsF2KzLgKIFVx45ETnMFYlAjAQEW_wy-

[39] “Discrete Event Simulation - an overview | ScienceDirect Topics.” Accessed: Nov. 05, 2024. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/discrete-event-simulation

[40] “What is Agile? | Atlassian.” Accessed: Nov. 05, 2024. [Online]. Available: https://www.atlassian.com/agile

[41] “Test-driven development - Wikipedia.” Accessed: Nov. 05, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Test-driven_development

[42] “What is behavior-driven development (BDD)? | Definition from TechTarget.” Accessed: Nov. 05, 2024. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/Behavior-driven-development-BDD

[43] “Design by Contract - an overview | ScienceDirect Topics.” Accessed: Nov. 05, 2024. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/design-by-contract

[44] “The Official YAML Web Site.” Accessed: Nov. 05, 2024. [Online]. Available: https://yaml.org/

[45] “PlantUML.” Accessed: Nov. 05, 2024. [Online]. Available: https://plantuml.com/

8. List of figures

Figure 1: PLC structure....12
Figure 2: PLC logic....14
Figure 3: Construction process....15
Figure 4: UML statechart nested states [9]....21
Figure 5: UML statechart orthogonal state [12]....21
Figure 6: UML statechart transition structure [9]....22
Figure 7: UML statechart initial state[9]....23
Figure 8: UML statechart initial state [9]....23
Figure 9: UML statechart choice pseudostate [9]....23
Figure 10: UML statechart junction pseudostate [9]....24
Figure 11: UML model of an ATM....26
Figure 12: UML statecharts of Release Controller and Machine Tool [29]....43
Figure 13: Layout of the line....48
Figure 14: Machine (M7) statechart....50
Figure 15: Controller (M7) statechart....51
Figure 16: New Controller (M7) statechart....54
Figure 17: New Machine (M7) statechart....55
Figure 18: New Buffer (B7) statechart....57
Figure 19: FIFO Function of the simulation....66
Figure 20: LIFO Function of the simulation....66
Figure 21: EDD Function of the simulation....66
Figure 22: CONWIP Function of the simulation....67
Figure 23: Statechart import command....67
Figure 24: Statechart interpreter command....67
Figure 25: Statechart bind command....68
Figure 26: Simulation queue....68
Figure 27: Clock initialization....69
Figure 28: Simulation time constraint....69
Figure 29: Sending enqueue event....69
Figure 30: Controller execution and simulation buffer update....69
Figure 31: Dispatching policy calculation....70
Figure 32: Queue for cycle initiation....70
Figure 33: Conwip policy calculation....70
Figure 34: Controller execution and dispatching policy calculation....71
Figure 35: Machine execution and remove from the queue called....71
Figure 36: Remove from the queue function....72
Figure 37: Failure start event reading....72
Figure 38: Failure finish event reading....72
Figure 39: Part arrival code....73

Figure 40: Buffer execution 74
Figure 41: Portion of the simulation output table 75
Figure 42: Plant Simulation Model layout 79
Figure 43: Part definition hierarchy 79
Figure 44: Part attribute 80
Figure 45: Input generation 80
Figure 46: Part generator settings 81
Figure 47: Buffer settings 82
Figure 48: Buffer exit method 83
Figure 49: Machine settings 84
Figure 50: Machine failure settings 84
Figure 51: Machine failure method 85
Figure 52: Final sink 85
Figure 53: Event Controller settings 85
Figure 54: Log structure 87
Figure 55: Log failure structure 88
Figure 56: Plant Simulation statistics report 88
Figure 57: KPI table 89
Figure 58: Calculate state time function 91
Figure 59: Machine state time calculation from Sismic output 91
Figure 60: Finished parts calculation 92
Figure 61: Reworked parts calculation 92
Figure 62: Machine state time calculation from Plant simulation log file 93
Figure 63: Unittest function output 94
Figure 64: Buffer validation function 94
Figure 65: File merging for comparison 95
Figure 66: Event removal after failure 96
Figure 67: File for comparison 96
Figure 68: State check results output 98
Figure 69: Events check results output 98
Figure 70: Buffer level checking cycle 99
Figure 71: Buffer kevel check results 100

9. List of tables

Table 1: Input objects parameters..101
Table 2: Input EDD parameters 102

Acknowledgement

We would like to extend our deepest gratitude to Professor Marcello Urgo, our advisor, for his invaluable guidance, expertise, and support throughout the course of this research. His insights into production systems and UML statechart applications have significantly shaped the direction and depth of our work. Professor Urgo's encouragement to think critically and explore innovative solutions empowered us to address complex challenges with confidence and rigor. His commitment to fostering a collaborative and enriching academic environment has left a lasting impact on our academic and professional growth. We are sincerely grateful for his mentorship and belief in our potential. Thank you for making this journey a truly rewarding experience.