Middleware for Ammunition Production Line

Advanced Software Architecture and Analysis

Project Report

Anne Lærke Kaihøj Siewartz Nielsen*, Jon Friis Jakobsen*, Mads Würtz Pedersen*,
Michael Ringhus Gertz*, Mikkel Bengtson Albrechtsen*,
University of Southern Denmark, SDU Software Engineering, Odense, Denmark
Email: * {annel21,jojak19,peder21,miger20,mialb21}@student.sdu.dk

Abstract—Her stal der stå et stykke abstrakt tekst.

This is a research project which aims to address the challenges of integrating systems in middleware within an Industry 4.0 environment. Using the case of developing an ammunition production line, the process entailed a systematic approach, utilizing appropriate tools to guide architectural decisions and system design, to ultimately render a solution that enables control of such production line. The paper covers how various tools (e.g. SysML, AADL, EAST-ADL, UPPAAL, CTL) and frameworks (such as Use Cases, Quality Attribute Scenarios (QAS), GQM) were utilized, along with a discussion of formal verification and validation (V&V) and formal design.

I. Introduction & Motivation

This paper reports on and examines the results of the task of building a new software middleware system for an industry 4.0 production line of (rifle) ammunition for sport and hunting. The middleware will be able to exchange and coordinate information to execute and change a production. The software is going to be continuously deployable and run 24/7.

The system uses sensors to measure ammunition along the production line, as well as being able to change the production process in real-time for different ammunition types.

The focus of the case is to integrate the sensors and machinery, with the central system already in place at the factory. Due to the type of production line, precision, safety, and performance are very important considerations for the system.

Moving forward, it is expected that the production line is scaled to multiple lines that run in parallel.

II. Problem, research question & approach

The current state of Industry 4.0 is challenged by the integration of systems. As uncovered by Jepsen et al. in [?], SDU's I4.0 lab is currently challenged with integrating the systems, as some systems are not ready for integration. This project therefore focuses on how a middleware system can be built for an I4.0 production line in the context of an ammunition production line. The middleware is abstracted away from the individual sensors and systems. It will therefore not focus on the currently lacking external interfaces of systems, but on how systems with integration options can be put together, using a middleware system. The project will also focus on the attributes needed for a production line, including integration, precision, safety, performance and scalability.

A. Research Questions

To ensure that our focus is placed correctly for our analysis, we use research questions that help clarify what task we need to solve:

1) What architectural patterns can achieve a seamless integration in a production line with middleware?
2) How can a system be designed to scale across multiple parallel production lines while maintaining performance and reliability?

B. Approach

To create a suitable I4.0 middleware system, a heavy focus will be put on the architecture of the system. Specifically, there will be a large focus on the quality attributes, as well as the interaction between elements. Identified requirements will be mapped to the quality attributes and tactics and patterns will be found to correctly tackle and fulfill these quality attributes. Feature modeling will be used to model the possible features, as well as their relation. ADL, AADL, EAST-ADL will be used to model the system to understand and evaluate it. UPPAAL will be used to model the states in the system, in order to evaluate if the behavior is possible. This consolidates into a formal validation and verification on the project. In order to ensure the system can be built in regards to its requirements, experiments will be conducted in critical areas.

III. Related work

In order to have a good starting point and build upon the existing knowledge space, it is important to first investigate what the state of the practice is in the field. Therefore, during this project, we have conducted a literature review to identify and review studies that relate to our research questions and topics. This can provide credibility to the claims we make and valid reasoning about the decisions we make.

We have largely utilized the primary study by Jepsen et al. [?] to build our knowledge and support our reasoning, as alluded to in Section II.

Studies by T. Borangiu and S. Răileanu [?] explain the role of middleware within cyber-physical systems for real-time monitoring of data integration. They recommend a decentralized system that does not limit the decision-making process but uses entities and channels that transport data and information.

IV. Use cases

Use case diagrams is a behavioral diagram (see Figure 16), that allows stakeholders to get a understanding of how the system is intended to behave. By creating use case diagrams we are able to define an overview of the process and formulate requirements.

In order to understand the case, a number of use cases have been elicited. These use cases serve as the project's backbone, as they list how the final product should function.

For this project, two main stakeholder groups have been identified: production managers, who are responsible for the production line and production orders, and machine operators, who handle the individual machines regarding maintenance and faults.

In order to produce the use cases several assumptions of the system have been made.

Part of a larger corporation, the factory is in charge of producing the corporation's offering of 2 different types of ammunition: Rifles and pistols. The factory has 5 production lines capable of producing all types of ammunition. The production facilities include 15 sensors and 5 different robots for each production line. The main challenge of this department is to integrate sensors and robots into a unified system that can integrate with the ERP system to dispatch productions.

The process of manufacturing is; to first get ammunition casing, then place ammunition casing on track. Get primer and insert primer into the ammunition casing. Get propellant and put in propellant into casing. Get ammunition head and insert ammunition head into ammunition casing. Get a package to pack ammunition into it and insert ammunition into the package, then close the package with ammunition in it and store packages in the warehouse. This results in a 12-step process. The moving of products to and from the production line is handled by a single system. Another system already exists for ordering material, the system to be built will therefore integrate with these systems.

Four use cases have been created. Use Case 01 (UC01) is the primary use case for the system, and includes production initialization. The use cases can be seen in the Appendix as Figure 15. Use cases for "Real-Time Monitoring and Control of Production Line", "Automated Quality Control and Defect Detection" and "Handling System Failures on the Production Line" have also been produced as use cases UC02, UC03 and UC04 respectively.

A. Requirements

Requirements are created by analyzing the description introduced in Sections I, II and IV, this is done in order to create the functional and non-functional requirements. These requirements will help to form the classes and describe what the expectations for the architecture is in form of quality attributes (QA).

Due to the project's structure of components, requirements for each of the different systems and their subsystems were created as individual entities. This division helped to maintain the overview and allow all systems to be developed individually without being dependent on each other.

A selection of the functional and non-functional requirements for the MES subsystem can be observed in Table I and II.

TABLE I: Functional Requirements for MES

ID	Functional Requirements
F1	Interface for receiving tasks for the production.
F2	Report back to the user if alerts in requested tasks.
F3	Retrieve data regarding production speed, machine status, and downtime.

TABLE II: Non-Functional Requirements for MES

ID	Non-Functional Requirements
NF1	Task determination to execution, must be delivered to the robot system within 500 ms.
NF2	Must scale in sync with the production line.
NF3	Must have an up-time of 99.999% and be available 24/7.
NF4	Must support continuous deployment, ensuring the system does not go down during software updates.

V. Quality attributes

Quality attributes are the desired traits of a given system. They are articulated as the non-functional requirements of an architecture. They help guide architectural design decisions to ensure that the legal and stakeholder non-functional requirements are met.

The quality attributes that will be the focus of this project are defined based on our research questions (see Section II-A), use cases (see Section IV) and the systems non-functional requirements (see Table II), and are as follows:

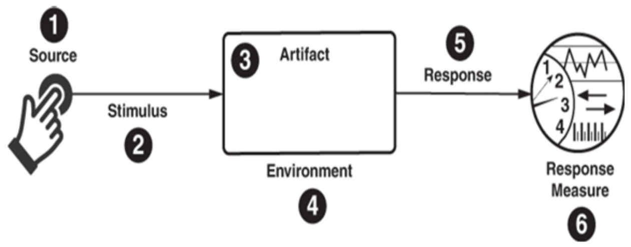
- Integrability the ability of the system to work together with other systems across existing or new systems.
- Modifiability is about the system being able to be changed quickly to handle requirements or errors that arise.
- Performance is the ability of the system to perform its functions quickly and efficiently.

- Availability is the system's ability to be operational and available to the processor.
- Scalability is the ability of the system to handle load increases without decreasing performance, or the possibility to rapidly increase the load.

A. Quality Attribute Scenarios

To better understand quality attributes, we describe them using Quality Attribute Scenarios (QAS). They describe the requirements in a systematic and verifiable manner that helps provide understanding for all stakeholders in the project.

Fig. 1: Quality Attribute Scenarios (QAS), consists of six elements that define the scenario



The process of defining QAS' is by defining the six elements, see Figure 1.

1) Source: who or what triggers the event
2) Stimulus: the type of event that occurs
3) Artifact: what part is affected
4) Environment: defines the conditions under which the event occurs
5) Response: the response to the event and
6) Response measure: how it is measured

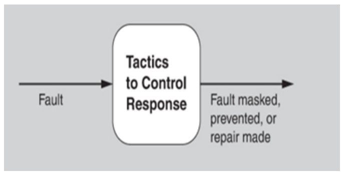
In this project, the following quality attribute scenario has been made for performance.

1) Source: 15 sensors, with an order of 1000 ammunition rounds being produced
2) Stimulus: Produces 300 sensor readings a second, total from all sensors
3) Artifact: MES system
4) Environment: Normal operations
5) Response: Process and track each reading and match it to a specific round
6) Response measure: Average processing time of 200 ms.

B. Architectural Patterns and Tactics

Tactics are the design decisions that explain how the quality attributes are used. The goal of a tactic can be the response to a stimulus by improving e.g. availability, see Figure 2.

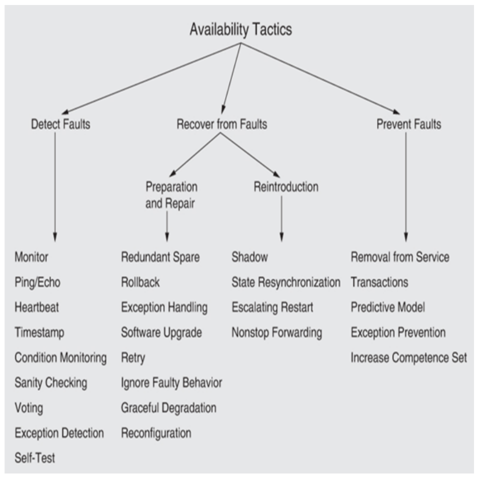
Fig. 2: Goal of Availability Tactics


Fault
Tactics to Control Response
Fault masked, prevented, or repair made

Tactics allow patterns to be adapted to a specific area, and by using a systematic process, it is possible to reinforce patterns using these tactics. Patterns focus on specific ways of implementing common software development solutions for common problems and do not directly link to quality attributes. Often, a single pattern may solve multiple tactics.

Figure 3 shows different tactics that can be used to improve availability, shown via a tree diagram, branching out according to the different problems that one desires to address.

Fig. 3: Availability Tactics



To get an overview of the architectural choices made for the different quality attributes, a questionnaire is created where questions are asked and they are recorded in a table. The answers to these questions can then be made the focus of further activities such as examining documentation, analyzing code or other artifacts, reverse engineering code, and so on.

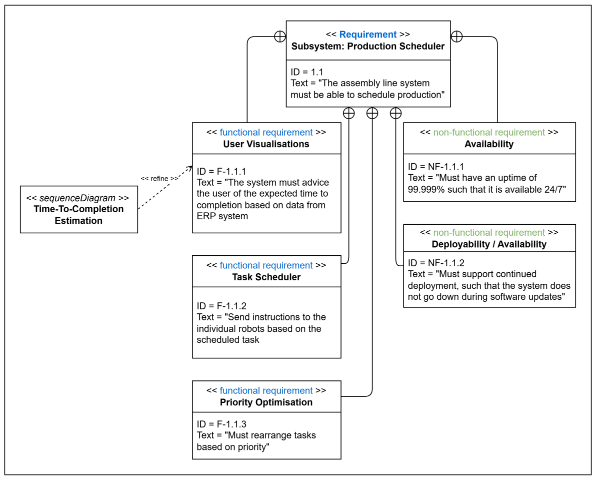
VI. Design and Analysis Modeling

After establishing the requirements for the system, the next step is to design the system. In this process, it is important to analyse the system, in order to create a solid foundation for the project. This section will therefore cover the design and analysis parts of the project, for the architecture and software.

A. Requirement Diagrams

Requirement diagrams is able to bridge the gap between text-based requirements while it helps improve requirement management throughout the system by providing traceability between text-based requirements and the model elements that represent the system analysis, design, implementation, and test cases.

Fig. 4: Requirement Diagram, giving a overview of the relations

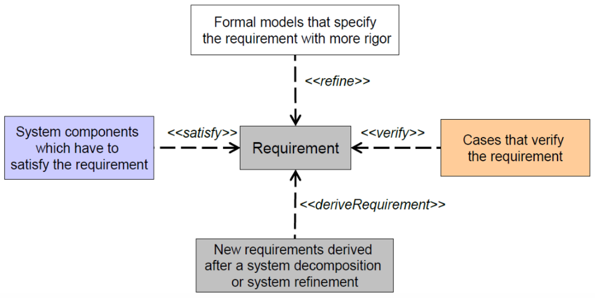


<< Requirement >>
Subsystem: Production Scheduler
ID = 1.1
Text = "The assembly line system must be able to schedule production"
<< functional requirement >>
User Visualisations
ID = F-1.1.1
Text = "The system must advice the user of the expected time to completion based on data from ERP system
<< refine >>
<< sequenceDiagram >>
Time-To-Completion Estimation
<< non-functional requirement >>
Availability
ID = NF-1.1.1
Text = "Must have an uptime of 99.999% such that it is available 24/7"
<< non-functional requirement >>
Deployability / Availability
ID = NF-1.1.2
Text = "Must support continued deployment, such that the system does not go down during software updates"
<< functional requirement >>
Task Scheduler
ID = F-1.1.2
Text = "Send instructions to the individual robots based on the scheduled task
<< functional requirement >>
Priority Optimisation
ID = F-1.1.3
Text = "Must rearrange tasks based on priority"

A requirement diagram, as seen in Figure 4, is a static structure diagram that shows the relationship between our requirements constructs, model elements that satisfies dependency, and test cases that verify dependency.

The purpose of requirement diagrams is to specify both functional and non-functional requirements in the model so that they can be traced to other model elements that satisfy them, and test cases that verify them. All while providing a hierarchy and traceability between requirements.

Fig. 5: Requirements Utilize Basic Relations, [?]



Formal models that specify the requirement with more rigor
<<refine>>
System components which have to satisfy the requirement
<<satisfy>>
Requirement
<<verify>>
Cases that verify the requirement
<<deriveRequirement>>
New requirements derived after a system decomposition or system refinement

As stated by Eun-Young Kang:

> The refine relationship in Figure 5 is shown with a dotted line with the keyword refine with the arrow pointing from the element representing the more precise representation to the element being refined.
> The satisfy relationship in Figure 5 is used to assert that a model element corresponding to the design or implementation satisfies a particular requirement. The actual proof that the assertion is true is established by the verification relationship. The satisfy relationship is shown by a dotted line with the keyword satisfy with the arrow pointing to the requirement to assert that the block satisfies the requirement.
> Trace relationship provides a general relationship between a requirement and any other model element. Useful for relating Reqs to source documentation or establishing a relationship between specifications. Shown as a dotted line with the keyword trace with the arrow pointing to the source document.
> Copy relationships support the reuse of Reqs by explicitly relating a copy of a Req to a source Req. The text property of the copied Req (slave Req) is a read-only copy of the text property of the source Req (master Req), but the copied Req has a different ID and may be contained in a different namespace. The copy relationship is shown by a dotted line with the keyword copy with the arrow pointing from the copied Req to the source Req.

B. Feature Modeling

A feature model is a systematic way to represent and manage different features in a software product. The model is primarily used to organize and visualize the system's features in a hierarchy, where each feature can be mandatory, optional, alternative, or part of a group of features. The purpose of a feature model is to help software developers understand and structure the common properties of the system and the variations that can be found between different system variants [?].

Fig. 6: Feature Model Overview of Production Line

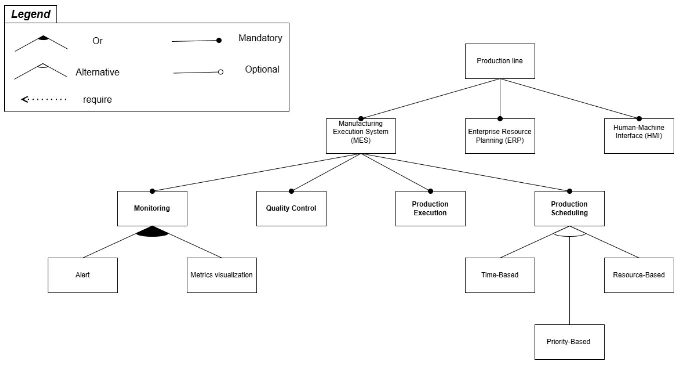


Figure 6 shows an overview of the production line with its system, subsystems and their features and sub-features. The production line is the system that branches into three main subsystems, Manufacturing Execution System (MES), Enterprise Resource Planning (ERP), and Human-Machine Interface (HMI). These three subsystems are mandatory, indicating that they are essential components for the operation of the production line in all configurations. The mandatory indicators are depicted by a line with a filled bubble, while optional indicators are lines with hollow bubble.

Monitoring is a subsystem that contains the features ”Alert”, and ”Metrics visualization”, which represents different ways of extraction metrics from the system. They are connected to monitoring with an OR relationship, allowing for either one or both of the features to be present in the system.

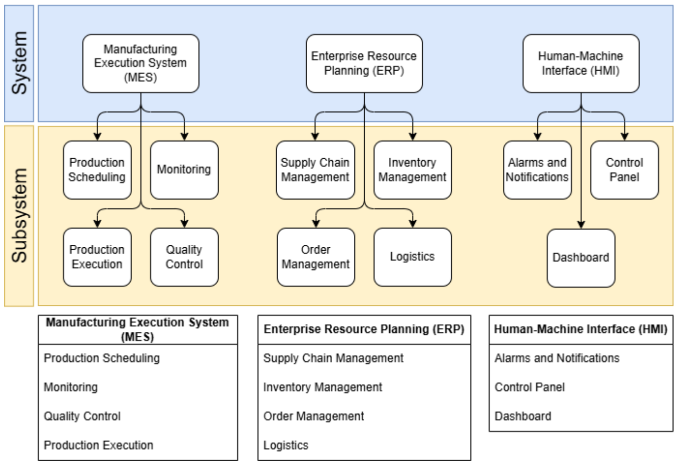
In the production scheduling module the feature description allow for alternative methods of scheduling. These alternatives are represented by a feature marker Time-Based, Resource-Based, and Priority-Based. The alternative marker makes allows for either one to be present, but only one.

From Figure 6 the require and excludes are not depicted. It is still important to know that the require relationships, indicates that one depends on the presence of another to function correctly. While the excludes indicate that two features or sub-features cannot coexist in the same configuration.

C. Systems and Subsystems

In order to facilitate the different operations the system is designed to, it has been identified that the system can be split into three different systems; Manufacturing Execution System (MES), Enterprise Resource Planning (ERP) and Human-Machine Interface (HMI). By doing this, the system is split into smaller systems, that different teams can work on. It also allows for the isolation of independent systems. By splitting it up, the final system is built as a micro-service architecture, as the individual elements are built and deployed independently of each other.

Fig. 7: A High-Level Diagram Depicting the Overall Architecture of the System



The splitting of systems, and identifying subsystems, also allows for better interoperability and inclusion of existing software solutions in parts of the final system.

In Figure 7 a diagram of the systems and subsystems can be seen. Each subsystem in the diagram may be made up of smaller modules and already existing software solutions.

With systems and subsystems identified, they serve as the base for the next step; the Architecture structure model.

D. Architecture Structure Model

In order to understand the system and be able to communicate about it, an architecture structure model can be made. The architecture structure model, is a model that represent the structure of the architecture. Being close to the Component & Connector (C&C) diagram type, architecture structure model is unique, in that it focuses on the high level structure of the application. The architecture structure model is an informal diagram type, and can be shaped to the level of details needed for the communication and documentation of the overall structure of the architecture.

In this project, the architecture structure model focuses on the individual systems, what elements are used and how they interact with other systems and pre-made elements. The model has allowed for discussion of the structure of the application, and how different systems interact. The architecture structure model has been split into three diagrams, which together form the full system. The diagrams are for each system in the project; HMI, ERP and MES.

Fig. 8: A structural diagram showing the structure and communication flow in the monitoring part of the MES subsystem

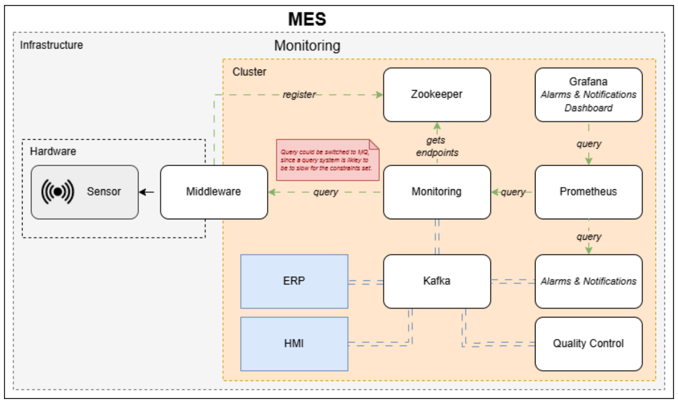


Fig. 9: A structural diagram showing the structure and communication flow in the execution part of the MES subsystem

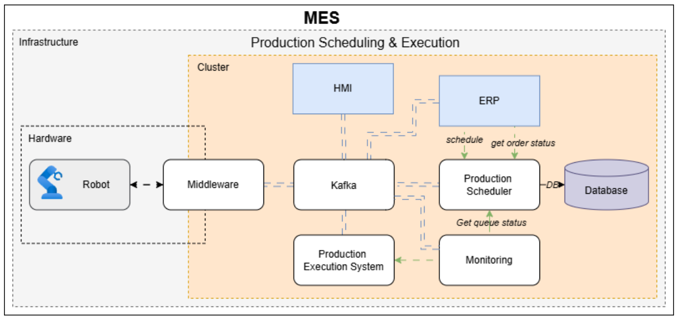


Figure 9 and 8 shows together the MES system, in regards to monitoring and production scheduling & execution. For the diagrams, blue boxes shows other systems, and white boxes shows subsystems to MES and incorporated existing software system, such as Kafka, Prometheus and Grafana. The diagram also focuses on the type of transport layer, with blue dotted lines representing queue connection through Kafka, and green representing REST connection with the arrow showing which way the connection were made. The combined, allows the communication of what elements the MES system is made of, including how existing software can be utilized in the system. Showing how the transport layer is laid out, allows for inspection of integration options, performance and scalability in the system. The diagrams also show the integration to physical elements, such as sensors and robots, and how communication with them will be handled.

The diagrams for ERP and HMI can be seen in Appendix on Figure 12 and 13 respectively. A full legend for the diagrams can be seen on Figure 14.

E. SysML

As stated by Eun-Young Kang:

> SysML is a graphical modeling language in response to the UML for systems engineering developed by the OMG (Object Management Group). Supports the specification, analysis, design, verification, and validation of systems that include software, hardware, data, and facilities. Intends to specify and architect systems and their components that can then be designed using other domain-specific languages (e.g. UML, AADL, EAST-ADL, etc.) for software design and 3rd party formal specifications/languages (i.e., timed-automata, promela, modelica, simulink, stateflow, etc.) for formal verification & validation (V&V). [?]

Represents:

- ”Structural composition, interconnection, and classification;” [?]
- ”Flow-based, message-based, and state-based behavior;” [?]
- ”Constraints on the physical and performance properties;” [?]
- ”Allocations between behavior, structure, and constraints;” [?]
- ”Requirements and their relationship to other requirements, design elements, and V&V cases.” [?]

F. AADL

AADL (Architecture Analysis and Design Language) is the process of describing a system, on multiple levels. This can be both hardware and software. AADL is also used across engineering disciplines to make sure all engineers on the project, understands the project and what the purpose of the project is.

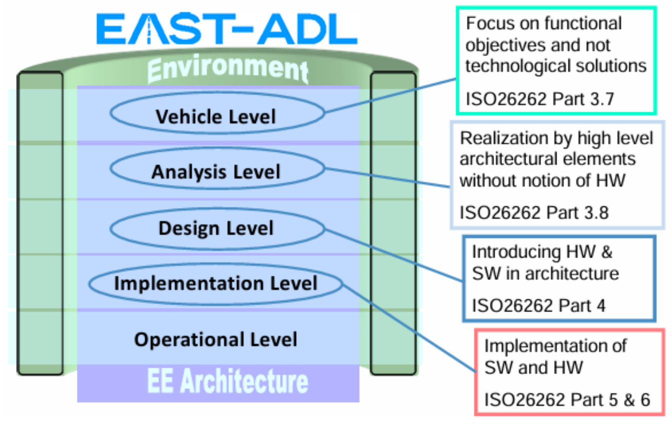
During the project AADL has been used to create a systematic description of the system, first a description of the overall system were written, then the system were broken down into three systems MES, ERP and HMI. Then each of these systems were defined into even smaller subsystems, see Figure 7. Each of these systems had their own responsibility which is then documented and described. Future work for this process, would be to split the subsystem into even smaller modules with their responsibilities. This is a process that would continue until all the system responsibilities has been accommodated.

The production line requires high availability with an expected uptime of 24/7, this can be handled with the help of a well-designed architecture. Therefore, by using AADL it is possible to gain insight into which components need high availability at any given time.

G. EAST-ADL

This is a guide that help with planning the process of the production line from start to end [?]. This is a process where each stage of the production is meticulously planned and executed.

Fig. 10: EAST-ADL Abstraction Levels


EAST-ADL
Environment
Vehicle Level
Analysis Level
Design Level
Implementation Level
Operational Level
EE Architecture
Focus on functional objectives and not technological solutions
ISO26262 Part 3.7
Realization by high level architectural elements without notion of HW
ISO26262 Part 3.8
Introducing HW & SW in architecture
ISO26262 Part 4
Implementation of SW and HW
ISO26262 Part 5 & 6

Using EAST-ADL, makes it possible to create abstraction at multiple levels, which allows for the design and definition of high-level production goals and refine them into detailed system implementations. By creating a functional decomposition with modular design, it is possible to reuse components across product lines, while integrated traceability ensures that design changes or errors can be easily tracked and resolved.

By using EAST-ADL we ensure that traceability and any errors or inefficiencies in our production line are identified and resolved as quickly as possible, which is important in safety-conscious industries like ours. EAST-ADL's focus on safety and reliability fits the requirement for the production line to operate safely in a 24/7 environment.

This process also enables the production line to be adapted to different types of ammunition or changes in the market. Furthermore, timing and real-time constraints ensure the synchronization and precision required for manufacturing processes, increasing overall efficiency.

1) Vehicle Level: This is the top level of the production line with high-level planning. At this level, the goals of the factory are defined, such as what types of ammunition will be produced, what quality standards will be met, and how the factory will operate safely and efficiently. This process is about understanding the big picture and ensuring the production line can produce different types of ammunition while maintaining accuracy and adhering to safety regulations.

2) Analysis Level: The analysis level is where the functions of the production line are identified and organized. This is about what the production line must do without considering specific machines or technology. The production line will use sensors to measure dimensions, machines to assemble components, and systems to perform quality control. Then each function is analyzed in detail, giving a blueprint of the tasks the production line must perform.

3) Design Level: The design level is the details of how these functions will be realized. Here, machines and tools are selected and their roles are defined. The conveyor belt for transporting cartridge cases, a robotic arm that applies the projectile, and sensors that provide quality checks are described. The production line layout is decided, ensuring that all machines and systems are correctly positioned and can communicate effectively with each other.

4) Implementation Level: The implementation level is the level where the production line is brought to life. The machines are installed, the software that controls them is developed and integrated, and the entire system is connected to ensure smooth operation. This could be the production scheduler to control the timing of each step in the production process, ensuring that materials move smoothly from one station to the next, and that errors are detected and corrected in real time.

H. Transport Layer

As the application is built into multiple smaller systems, integrating each system is crucial. A transport layer is introduced to facilitate the transfer of data between each part of the system.

As the system is very data-oriented, in the form of pulling data from sensors and pushing commands to machines, as well as representing the current actions and states in an HMI, an efficient transport layer is needed. This is especially highlighted by the requirements, which state precision, performance and scalability, as well as availability, is required.

With these aspects, two different approaches to the transport layer are used: Pull-based and push-based. Pull-based is used when an application needs to retrieve data from another application, and it is in charge of ensuring that it gets the data. In this system, REST is used as the format for pull-based requests. Soap, GraphQL, gRPC and similar are also viable options and may be introduced by third-party systems. However, providing a uniform API format across the system allows for easier integrations. The use of REST is also most common in the industry, but gRPC may be used where performance and data format are crucial. Therefore REST is the standard in the application, but if critical time-sensitive components are identified in the development phase, they may use gRPC, but the standardized format is REST for its simplicity and ease of use for external- and web-facing services.

Pull-based is not suitable for the whole of the system. As seen in section VI-D, push-based transport is needed. Push-based communication is based on the notion, that an application wants to push data to other parts of the application that need the data, whereas pull-based based is where applications reach out themself for the data or to trigger actions. Push-based can be incorporated in three different ways; queues, webhooks and websockets. However, due to the scale of components that need to interact, queue-based communication is suitable. Queue-based communication allows for a broker to be used. By using a broker, applications that push data do not need to know about whom needs the data, and applications can listen on any queue they want, through the broker. Queues also allow for the queuing of messages, while previous data is processed.

Queues therefore provide a scalable system for communicating large volumes of data, using push-based methodologies.

In the system, Kafka has been used as the queue service and broker. Alternatives such as RabbitMQ exist, but Kafka is suitable in this situation, as it is built for high availability, scalability and performance in scaled-up environments.

I. Containerization

Containerization is a technology that involves packaging an application and its dependencies into an isolated and self-contained container.

Containerization is an important part of the project, as it allows for enhanced deployability. Through the use of Docker as a containerization engine, it is possible to develop and ship software solutions, that run on a larger number of machines, without having to fine-tune each machine to run the software. Orchestration tools like Kubernetes then allow for large-scale deployments, that can live up to stringent availability requirements.

With the use of Docker and an orchestration tool like Kubernetes, the software is intended to be deployed on multiple machines simultaneously and ensure the software is available 24/7.

J. Programming Language

Choosing the most suitable language for each subsystem ensures that the system is efficient, easy to maintain and responsive in real-time operations.

1) Real-time sensor data processing and machine control: C++ is highly efficient and works well for applications that require fast response times and precision. It is particularly suitable for handling sensor data and controlling machines on the production line, where low latency is essential for detecting defects and making adjustments in real time.

2) Middleware and System Integration (ERP, MES): Java is a reliable choice for enterprise applications due to its performance, cross-platform compatibility and scalability. It supports the high throughput requirements of ERP and MES systems, making it ideal for integrating and managing production schedules and inventory in real time.

3) Quality control and defect detection: Python is widely used in machine learning and data analysis, making it ideal for tasks such as defect detection and quality control. Libraries such as OpenCV and TensorFlow simplify the development of AI algorithms for high-accuracy inspections.

4) Reporting and real-time dashboards (HMI): JavaScript (Node.js and React), with frameworks like React for the frontend and Node.js for the backend, is perfect for building interactive dashboards and real-time monitoring tools. This setup allows operators to easily view production data, alerts and reports as they happen, ensuring an efficient user experience on the Human-Machine Interface (HMI).

K. Databases

The system uses three different kinds of databases; Time-series, relational SQL and document-based NoSQL.

A time-series database is chosen for data that changes in one axis over time, such as sensors. Both SQL and NoSQL is not suitable for time-series, as when data frequency and number of total datapoints stored, reach a certain point, the query time would be too long. Time-series databases, such as InfluxDB and Prometheus, are specifically built for the storage of data streams from sensors and similar.

Relational SQL is used for core elements, such as scheduling, ordering and similar activities where there is a focus on transactional data. For this, databases such as PostgreSQL and MariaDB are often used.

Document-based NoSQL is used, to store larger documents, such as images from image sensors. Document-based NoSQL can also be used to store logs from the system itself. For sensor data, that does not fit in time-series, a database like MongoDB would fit, whereas for logs a database like Elasticsearch is more optimal. However Elasticsearch has a larger focus on search, and image sensor data would be possible to store in Elasticsearch as well, eliminating the requirement for a database like MongoDB.

VII. Formal Validation and Verification (V&V)

The validation and verification of an architecture is important to ensure proper evaluation of the architecture. In this chapter we will discuss when to test an architecture along with some tools to formally state and test the logic of a system.

A. When to Test Architecture

Evaluation of an architecture is vital to mitigate risks and better ensure project success.

There are some significant points throughout the software development cycle where an evaluation of the architecture can prove crucial. Early in the process, when the requirements are not yet fully locked in, a discovery review can be conducted with stakeholders with decision power. This review can be an important help to ensure the architecture being developed are addressing the required quality attributes to satisfy customer needs. Later in the development process, when developers begin to make decisions that depend on and are influenced by the system architecture, an evaluation can be held to determine what decisions should be made. This is done when the costs of reverting those decisions outweighs the costs of conducting the evaluation. Frameworks for evaluating software in these stages include Architecture Tradeoff Analysis Method (ATAM) for early evaluation to identify business goals and elicit QA requirements and sets of risks, sensitivity- and tradeoff points. Additionally Lightweight Architecture Evaluation (LAE) can more regularly be conducted low-cost and less formally as internal peer reviews, using elements of the ATAM to evaluate changes since the last review. Very late, when the system has largely been implemented, the architecture can be tested and measured to determine whether the architecture fulfills the desired quality attributes. [?]

B. UPPAAL

UPPAAL is a model checker for modeling, validation, and verification of real-time systems. It works using a collection of what is called ”networked timed automata” which can be thought of as state-machine diagrams that can communicate with each other. To do model-checking in UPPAAL one needs two things, a model and a requirement specification.

1) Model: The model is a collection of timed and networked automata. As stated above, an automaton (plural: Automata) is based on the idea of a state-machine diagram. It uses a graph-style layout with nodes and edges. These automata describe the behavior of the system.

2) Requirement specification: The requirement specification has two parts which are described by CTL notation (described in section VII-C)

- Invariant (aka. safety property): the requirement that something (bad) never happens. This is often a deadlock where there are no edges to take from a given node so the model is stuck in the current state. But could also be mutual exclusion or and error state.
- Liveness: the requirement that something (good) will eventually happen, e.g. a specific state is achieved.

C. Computation Tree Logic

Computation Tree Logic (CTL) can be seen as a language for asking questions regarding a system. You can use CTL to ask ”What will happen in the future”, or ”Is something always true”. The system can be both hardware, software or a combination. CTL is used to describe what a component does and doesn’t.

$$AX(red \rightarrow green)$$

1) CTL Example: Imagine a traffic light system managing the north-south direction. It can have two possible states:

- NS_red: The light is red for north-south traffic.
- NS_green: The light is green for north-south traffic.

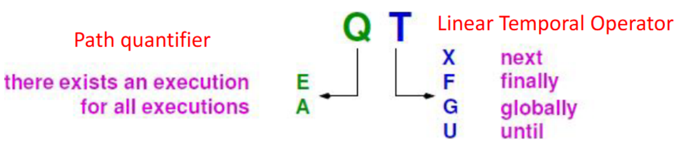
We want to ensure that ”The north-south light must remain red until it turns green”. The CTL Formula for this CTl is

$$E(NS\_red \cup NS\_green)$$

Explanation:

- NS_red must remain true as long as the light has not turned green.
- The system will eventually reach a state where NS_green is true, and the requirement stops.

Fig. 11: An explainer for path qualifiers and linear temporal operators in CTL



A CTL statement is build up by Path Quantifier and Linear Temporal Operators. We always need to have at least one Path Quantifier and one Linear Temporal Operator.

2) Path qualifier (PQ): Path qualifiers are used to describe the scope of the CTL-formula.

- E: There exists at least one path
- A: For all possible paths

3) Liniar Temporal Operators (LTO): Liniar temporal operators describe in what scenario the following statement needs to be true

- X: Next - true in next state
- F: Finally - true in any future state
- G: Globally - true in all future states
- U: Until - true until a different state is true

How CTL works: A system can have multiple paths or scenarios (lets call them roads), imagine.

> You standing at a road, and there is multiple intersection in front of you.
> Each road represents a future state of the system.

With CTL you can ask some of the following questions.

> ”Can I always go left?”
> ”Can I get to my destination by one or another read?”
> ”What happens, no matter which road I take?”

VIII. Evaluation

To evaluate the process we have been through, it is important to analyze whether the requirements and goals that have been defined have been answered throughout the process. It is important to evaluate in the areas that have been defined based on our research questions, which are our focus area. The following experiment is a experiment answering the research questions in II-A

A. System Experimentation

Due to a production line that demands quality attributes such as performance, integrability, modifiability and availability, it is necessary to validate whether the sensor data generated can reach the modules that need the data, efficiently and reliably to act. Therefore, the experiment is built around Kafka’s ability to transport messages across the system quickly and reliably and with different amounts of sensors due to the requirement to be able to scale.

In order to conduct a test, the Goal-Question-Metric (GQM) framework was used.

Goal: production efficiently precess data in real-time.

Questions:

- How quickly can anomalies be detected and communicated to Production Environment systems (PES)?
- How scalable is the solution for multiple production lines?

Metrics: Latency (milliseconds), throughput (events/sec), and anomaly detection accuracy.

The method for carrying out the experiment is based on creating a pipeline where real-time sensor data is streamed and collected by modules that use this data.

In order to evaluate this, the following hypotheses are made.

Low Latency Hypothesis

- H_0: Kafka cannot maintain low latency ($< 5\,\mathrm{ms}$) for real-time data processing under industrial-scale workloads.
- H_1: Kafka can maintain low latency ($< 5\,\mathrm{ms}$) for real-time data processing under industrial-scale workloads.

Scalability Hypothesis:

- H_0: Kafka's performance deteriorates significantly as the volume of data streams increases (e.g., latency exceeding $10\,\mathrm{ms}$).
- H_1: Kafka's performance scales efficiently with increasing data volumes, maintaining consistent latency and throughput.

The produced data was saved in CSV files to be analyzed using various graphs. The data was collected by running the experiment five times with different workload levels where the amount of events/per sec was increased.

However, there are also limitations that affect the validity of the experiment, as the experiment was conducted in a simulated environment, which therefore cannot be compared to how it would function in a real-time environment.

The result showed that it was possible to prove the given hypotheses, since it was possible to have a latency ($< 5\,\mathrm{ms}$) and high throughput ($> 1{,}000$ events/sec). Therefore, it is possible for Kafka to comply with scalability, reliability and safety as it can send alarm messages across the system within ($< 5\,\mathrm{ms}$).

Future work will explore scalability across multi-line production setups, improve latency under higher data loads, and evaluate performance in real-world industrial environments.

IX. Discussion

This section will highlight some key aspects of Formal V&V, why it is used and when, as well as the use of experimental evaluation. At last, it will also highlight why formal design is necessary.

A. The Importance of Formal V&V Before System Implementation

Developing a larger system is often a lengthy and expensive process. By doing V&V it is possible to reduce the risk, that a large project introduces, in regard to its design and structure. By using methods such as UPPAAL, it is possible to validate that processes are possible and optimal. By mapping and modelling system architecture and components in its different forms, blind spots are uncovered and can be corrected.

B. Comparing Formal V&V and Experimental Evaluation

Formal V&V is different from experimental evaluation, as formal V&V is done theoretically and in simulations. Therefore, no real code that can be used is produced. The system in evaluation is put in a vacuum per se. The structure of the system is evaluated, and the behaviour is modelled. In experimental evaluation, the focus is on real-world experimentation, where code is being written. However, the code may be limited and restricted. It allows us to test real software issues and see if frameworks and areas that are less understood for the project, are to be explored. Experimental evaluation therefore focuses on a closer example to the real software and tests the hypothesis created for the system to be built or being built. However, Experimental evaluation can be time-consuming, and not be able to mimic all aspects of the final product, as it will always be a subpart of the final product. Experimental evaluation may also use methods such as Software-in-the-loop (SIL) or Hardware-in-the-loop (HIL), which allows the built software to be tested against simulated environments (either through software simulation, SIL, or through hardware simulation, HIL). This allows for code to be tested against known scenarios, and test to see if the software can handle different scenarios. However, these methods are used in the development phase, to faster test software against complex scenarios, but could be used in the evaluation of architecture on small-scale prototype code.

The two methods therefore represent two sides of validation, as one focuses on theory and simulations, and the other focuses on a closer to reality, but more time-consuming and limited testing.

C. Necessities of Formal Design and Analysis and Its Benefits

Before moving to experimental evaluations, either using prototype code and testing the hypothesis of the architecture or by using SIL and HIL methods, formal design an analysis, as well as V&V is a crucial step. The design, analysis and V&V provide a base, in which the architecture and software can be built. Ensuring the software is well designed, and done so on an analysis, allows for the architecture to be well structured and provide a good development life cycle. Without a proper design, built on formal designs and analysis, the software may be built incorrectly, which could end up costing a large sum and time to correct at a later stage. Therefore, doing design early, and on an objective basis, allows for catching errors early and correcting them while it is possible without impacting the total cost and timeline of the project.

X. Conclusion

The project involved creating a system architecture for an ammunition production line, emphasizing the use of appropriate tools to execute the project systematically and in a structured manner. Based on the research questions posed, a literature review was conducted to examine previous studies in the field. The focus was on addressing the research questions defined in Section II-A by analyzing insights from related studies. Throughout the process, the questions were answered using a systematic architectural model and experiments, which demonstrated the system's ability to scale while maintaining low latency and high availability in the simulated environment. However, further work is required to validate the architecture in a real-time environment.

References

[1] J. Wan, S. Tang, D. Li, M. Imran, C. Zhang, C. Liu, and Z. Pang, "Reconfigurable smart factory for drug packing in healthcare industry 4.0," IEEE Transactions on Industrial Informatics, vol. 15, no. 1, pp. 507–516, 2019.

[2] Y. Alsafi and V. Vyatkin, "Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing," Robotics and Computer-Integrated Manufacturing, vol. 26, no. 4, pp. 381–391, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736584509001239

[3] P. Leitão, J. Barbosa, A. Pereira, J. Barata, and A. W. Colombo, "Specification of the perform architecture for the seamless production system reconfiguration," in IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society. Florence, Italy: IEEE, 2016, pp. 5729–5734.

[4] G. Angione, J. Barbosa, F. Gosewehr, P. Leitão, D. Massa, J. Matos, R. S. Peres, A. D. Rocha, and J. Wermann, "Integration and deployment of a distributed and pluggable industrial architecture for the perform project," Procedia Manufacturing, vol. 11, pp. 896–904, 2017, 27th International Conference on Flexible Automation and Intelligent Manufacturing, FAIM2017, 27-30 June 2017, Modena, Italy. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978917304018

[5] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, "Reconfigurable manufacturing systems," CIRP Annals, vol. 48, no. 2, pp. 527–540, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850607632326

[6] Y. Koren and M. Shpitalni, "Design of reconfigurable manufacturing systems," Journal of Manufacturing Systems, vol. 29, no. 4, pp. 130–141, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612511000021

[7] M. Bortolini, F. G. Galizia, and C. Mora, "Reconfigurable manufacturing systems: Literature review and research trend," Journal of Manufacturing Systems, vol. 49, pp. 93–106, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612518303650

XI. Appendix

Fig. 12: A structural diagram showing the structure and communication flow in the ERP subsystem

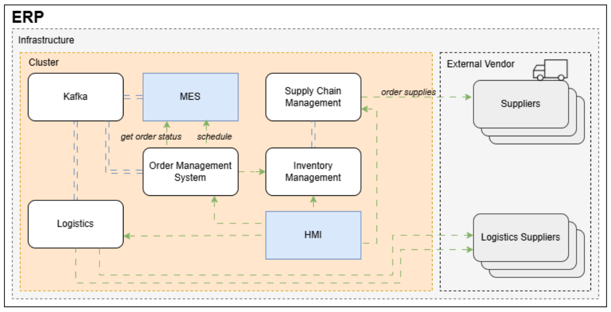


Fig. 13: A structural diagram showing the structure and communication flow in the HMI subsystem

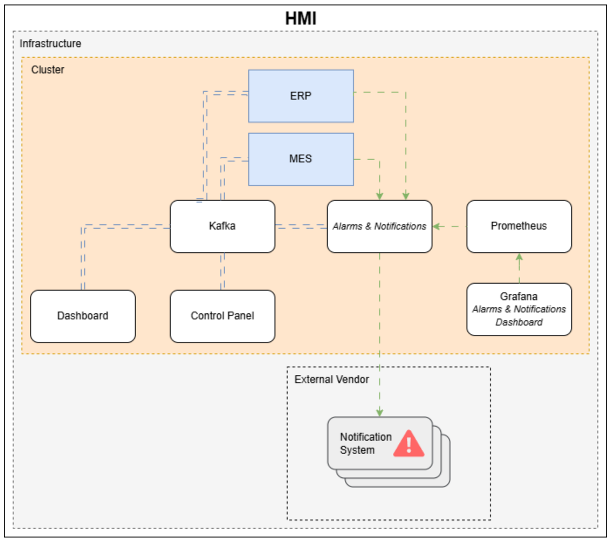


Fig. 14: The Legend for All Structural Diagrams

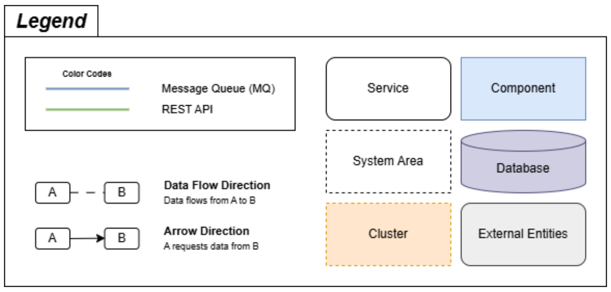
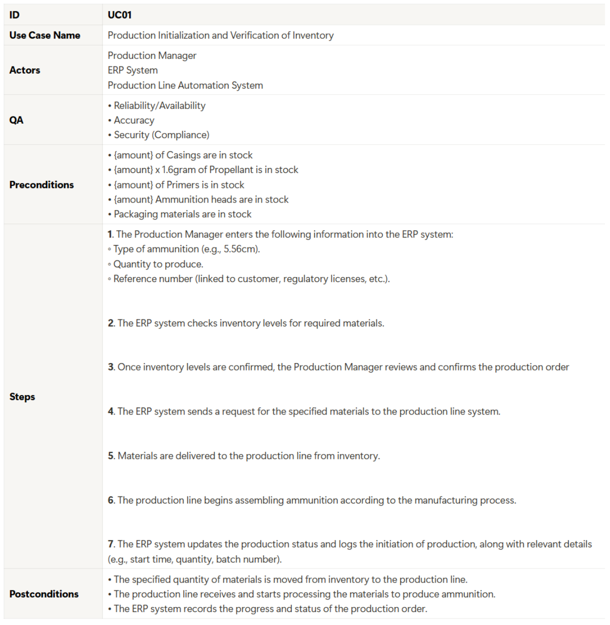


Fig. 15: Use Case UC01



ID	**UC01**
Use Case Name	Production Initialization and Verification of Inventory
Actors	Production Manager ERP System Production Line Automation System
QA	• Reliability/Availability • Accuracy • Security (Compliance)
Preconditions	• {amount} of Casings are in stock • {amount} x 1.6gram of Propellant is in stock • {amount} of Primers is in stock • {amount} Ammunition heads are in stock • Packaging materials are in stock
Steps	**1**. The Production Manager enters the following information into the ERP system: ◦ Type of ammunition (e.g., 5.56cm). ◦ Quantity to produce. ◦ Reference number (linked to customer, regulatory licenses, etc.). **2**. The ERP system checks inventory levels for required materials. **3**. Once inventory levels are confirmed, the Production Manager reviews and confirms the production order **4**. The ERP system sends a request for the specified materials to the production line system. **5**. Materials are delivered to the production line from inventory. **6**. The production line begins assembling ammunition according to the manufacturing process. **7**. The ERP system updates the production status and logs the initiation of production, along with relevant details (e.g., start time, quantity, batch number).
Postconditions	• The specified quantity of materials is moved from inventory to the production line. • The production line receives and starts processing the materials to produce ammunition. • The ERP system records the progress and status of the production order.

Fig. 16: Structural Diagram - Overview of Different Diagrams

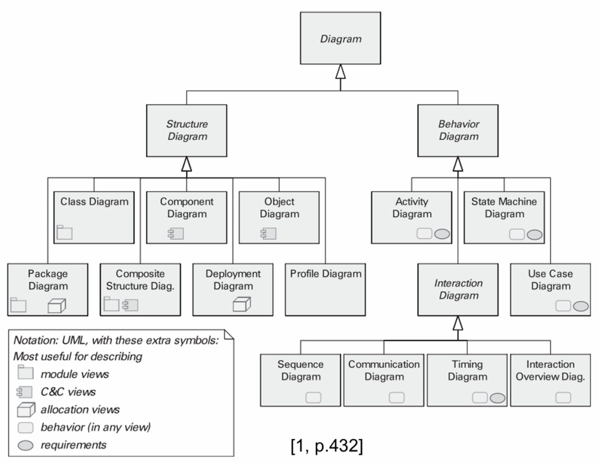


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
State Machine Diagram
Package Diagram
Composite Structure Diag.
Deployment Diagram
Profile Diagram
Interaction Diagram
Use Case Diagram
Notation: UML, with these extra symbols:
Most useful for describing
module views
C&C views
allocation views
behavior (in any view)
requirements
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diag.
[1, p.432]