Group Report Template

Anne-Marie Rommerdahl*, Jannatul Ferdous†, Umma Soneyatul Jannat‡, Student 4§
University of Southern Denmark, SDU Software Engineering, Odense, Denmark
Email: * {anrom,umjan25,student4,student5}@mmmi.sdu.dk

***Abstract*—**

***Index Terms*—Keyword1, Keyword2, Keyword3, Keyword4, Keyword5**

I. Introduction and Motivation

The structure of the paper is as follows. Section **??** outlines the research question and the research approach to analyze the research question and evaluate our results. Section **??** describes similar work in the field and how our contribution fits the field. Section **??** presents a production reconfiguration use case. The use case serves as input to specify a reconfigurability QA requirement in Section **??**. Section **??** introduces the proposed reconfigurable middleware software architecture design. Section **??** evaluates the proposed middleware on realistic equipment in the I4.0 lab and analyzes the results against the stated QA requirement.

II. Problem and Approach

Problem. In todays society, there is a heavy focus on the automation and digitalization of complex systems. The complex systems are used in many areas, such as industry, healthcare, public transport and so on. These systems are described as complex, because of the many parts involved, the intricate ways in which those parts communicate, and the overall behavior of the system - how do all these parts work together to complete the systems' goals? To ensure the correctness and reliability of such systems, it is of outmost importance that thought is put into the design. After all, failure in a system can lead to negative consequences ranging from minor inconviniences to death[SOURCE Lecture 1].
The Industry 4.0 production (I4.0) domain is characterized by the integration of physical components with different kinds of technology and the communication channels between. In this domain there exist many complex production systems, one of which is the production of cars. Cars are sold not only for their functionality as a vehicle of transport, but often offer the customers a range of options for customization. These range from functional (such as the engine) to cosmetic (such as the color). Building a system that can handle customer orders and organize production of the placed orders is no small feat. It is one thing to design an architecture for such a system, but how does one ensure that it fulfills the requirements? Changes to the system during later parts of the development process are much more costly than changes made earlier in the process. Therefore it is important to ensure that the architecture is designed well, before implementation starts. For a system which customers directly interract with, an important aspect is the performance. Not only must the system provide quick feedback to customer actions, it must also be able to do so while servicing multiple customers at the same time. From this, we've created these research questions: *Research questions:*

1) How to design software architecture for performance?
2) How to evaluate the software architecture in regards to performance?
3) How well does a prototype built based on this architecture perform perfomance-wise?

Approach. The following steps are taken to answer this paper's research questions:

1) smth about each section of the report? Like, use cases were made to get an idea of what the system should be able to do?

III. Related work

This Section addresses existing contributions by examining articles about software architectures for performance in the I4.0 domain. In total, x papers are investigated.

The paper [**?**] [evaluates performance to improve system but only at run-time.] [**?**] [case study implementation of a Traffic Management System to evaluate performance of several architectures.]

[**?**][This study proposes a model-driven approach by transforming software architecture into performance models, allowing performance to be analyzed early in the development process.]

[**?**][The authors show how performance models can be systematically constructed from architectural descriptions that include performance-related details, enabling early evaluation of system performance before implementation.]

[**?**][The paper addresses the performance quality attribute by systematically examining how software architectures specify, analyze, and evaluate performance requirements such as latency, throughput, and response time.]

While these contributions provide knowledge and new ideas about performance in the I4.0 domain, there is little to no focus on studying performance as a quantifiable architectural requirement, and how to evaluate a software architecture that adopts this requirement.

In [**?**], experiences are elaborated on a three-layer architecture of a reconfigurable smart factory for drug packing in healthcare I4.0.

The paper [**?**] proposes an ontology agent-based architecture for inferring new configurations to adapt to changes in manufacturing requirements and/or environment.

In [**?**], [**?**] an architecture for a reconfigurable production system is specified. Two objectives for reconfiguration and how they can be reached are described.

Several papers [**?**], [**?**], [**?**] describe reconfigurable manufacturing systems that are cost-effective and responsive to market changes.

All contributions provide valuable knowledge about reconfiguration but lack a study of the software architecture perspective that specifies a quantifiable reconfigurability architectural requirement, a software architecture that adopts the architectural requirements, and evaluates the architectural requirement.

IV. Use Case

This Section introduces the use case of a customer placing an order on the company's website. This use case covers the sequence of actions starting from the customer accessing the website, to the moment when the order is saved and marked as 'ready' for production in the system.

A. Customer places an order

Actors: Customer, Website, Production Scheduler Preconditions: Available cars and customization options have been defined in database. Steps:

- Customer accesses company website
- Customer chooses preferred car
- Website fetches available customization options for chosen car
- Customer chooses preferred customizations
- Customer places the order
- Website sends order details to the Scheduler
- Production Scheduler recieves order details from website
- Production Scheduler breaks down the order into a JSON production recipe, which is then stored in the database
- Production Scheduler stores the order in the database, marked as 'ready'

Postconditions: A customer order is stored and marked as 'ready' for production.

V. Quality Attribute Scenario

This section contains the quality attributes for the architecture and also contains one or more Quality Attribute Scenarios (QAS), which illustrate the system's scenario.

A. Performance

This section contains the quality attributes for the architecture and also contains one or more Quality Attribute Scenarios (QAS), which illustrate the system's scenario.

B. Performance quality attribute

The performance of the system starts from the moment a customer has placed an order on the website, and the performance depends on the website, Non/Non-relational DB, Kafka Message bus, and Production Scheduler. Communication between these systems must take place within ten units.

Scenarios: Two scenarios have been made for Performance.

The first scenario on **??** shows that when a customer places an order through the car-selling website, it will publish to Kafka that an order has been created. After this event is done, the production scheduler fetches the message and starts generating the JSON production recipe, and marks it as ready as soon as it is stored in the database. This entire event chain will be completed in 10 time units. The customer's order submission to Kafka to get the message; this process must complete within four milliseconds. This scenario ensures that distributed communication does not cause significant latency and captures the system's performance requirement for the major workflow.

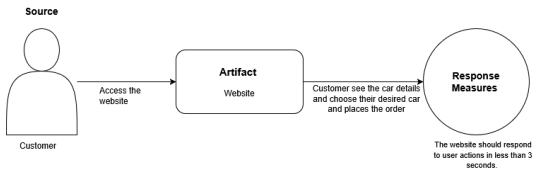

Fig. 1: Scenario 1 for the website

The second scenario on **??** shows that after the Kafka message bus receives the order from the website, the scheduler must start immediately to prevent delays. In normal load conditions, as soon as the message is available in Kafka, the scheduler must start generating the recipe within three milliseconds. This prevents performance bottlenecks in the message queue and also ensures that the event-driven architecture facilitates quick transitions from order creation to recipe development.

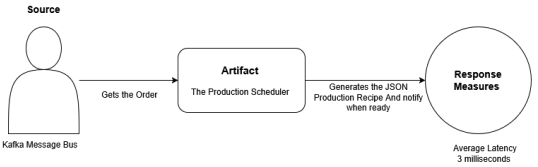

Fig. 2: Scenario 2 for the Scheduler after customer places the order

The scenario illustrated in **??** and **??** describes the performance of the system. The process begins when the customer places the order through the website and, after that, the scheduler gets the order from the website through the Kafka message bus. The production scheduler then breaks the order into a JSON recipe and stores it in the database, marking it 'ready' after storing it in the database. The complete event chain—from order submission to recipe persistence—is depicted in **??** and **??**.

C. Requirements

Based on the QAS and the use case, several requirements were defined, which can be seen in table **??**[FIX TABLE LABEL!!].

Req ID	Description	Type
RQ1	Customer can choose from a list of available cars	Functional
RQ2	Customer should be able to chose a car and customize the car's model, engine and color	Functional
RQ3	Scheduler should prepare a placed order for production by creating and saving a production recipe	Functional
RQ4	Scheduler should keep track of an order's status (ready, in progress, completed)	Functional
RQ5	The website should respond to customer placing an order in less than 3 seconds	Non-Functional
RQ6	Any order placed on the website should be saved and marked as 'ready' for production in database within 10 time units	Non-Functional

D. Tactics for Quality Attribute

To ensure the health of the connection from the systems, the most fundamental tactic is event-based processing, which separates the Website from the Scheduler and guarantees that user-facing operations do not hinder generating the JSON production recipe or database writing. It also covers the bound execution times tactics, as after the customer places the order, the message bus will receive the message within three milliseconds, and the Production Scheduler will generate the recipe within three milliseconds. To reduce overall latency, the system additionally relies on concurrency by simultaneously allowing the website's backend to deploy uWSGI with multiple workers to handle several HTTP requests simultaneously. Each worker runs an instance of the Django application and processes user interactions independently. By doing so, it prevents new orders from being blocked and ensures the website remains responsive under concurrent user loads. Overall, concurrency improves throughput and response time without requiring multiple deployment replicas.

VI. Design and Analysis Modelling

This section describes the different steps in the design process, as well as an analysis of the design.

A. Features

Figure **??** shows the features for the car production system that will realize the defined requirements. The system will have a website, from which the customer may browse (RQ1) and choose a car to customize (RQ2). The Scheduler will process the orders by creating a production recipe for each order (RQ3). The Scheduler also manages the status of each order (RQ4), setting status to 'ready' for production when the corresponding production recipe has been created and stored in the system.

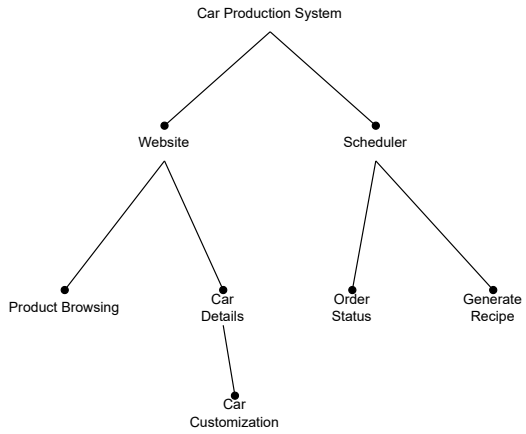

Fig. 3: Feature Diagram of the car production system

B. Structural Analysis Model

A structural analysis model (also called smth FAA) can be used to present the functions for the features defined in figure **??**, and the way data flows between the different parts. Figure **??** shows the FAA of the Scheduler.

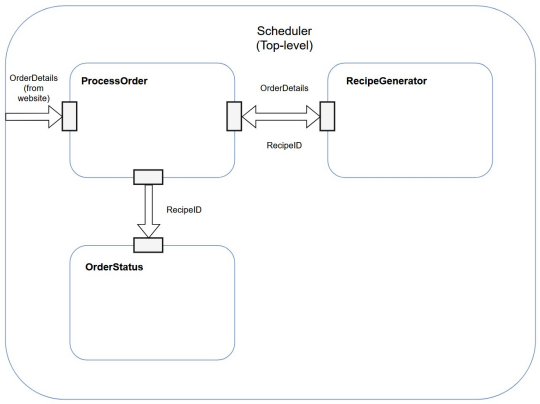

Fig. 4: Functional Analysis Model of Scheduler

C. Finite-State Machine

To show the behavior of the system, finite-state machines (FSM) were defined for the website and scheduler, shown in figure **??** and **??** respectively. When accessing the website, all available products are shown to the user. The user may navigate to the sub-page containing the details for a specific car, where they may customize the car and add it to their cart. Adding it to their cart will redirect them to the cart, where they can checkout to place their order. Finally, they will be shown a confirmation screen. From any page on the website, they directly return to the homepage. When the scheduler starts, it will listen for messages from the website. Once a message has been received containing details of an order, the scheduler will process the order, before returning to listening.

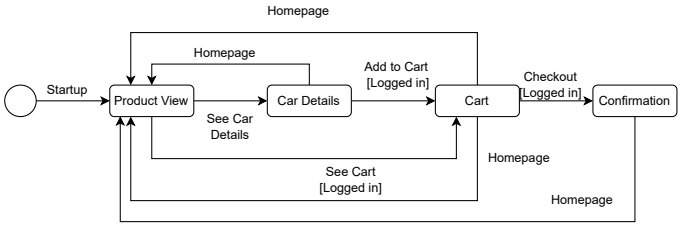

Fig. 5: Finite State Machine for the website

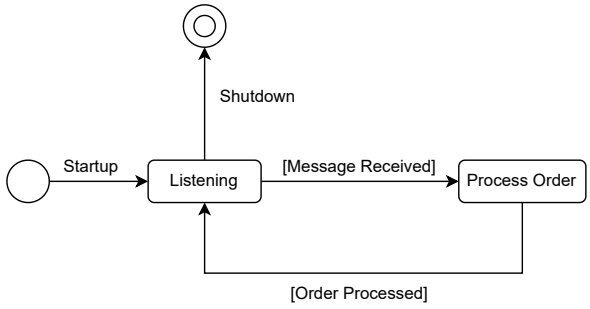

Fig. 6: Finite State Machine for the scheduler

D. Systems and subsystems

The architectural design follows a distributed system composed of multiple subsystems, with each one responsible for a specific part of the overall functionality. The system is structured to support quality attribute scenarios defined in Section **??**.

The system consists of three main subsystems: a user interface subsystem, a production scheduler, and a communication subsystem that enables event-driven communication between them. This section below will discuss the technology that is used for the subsystems.

1) Message bus Apache Kafka:: To establish communication between the website and the production scheduler, Apache Kafka will be used. The architecture requires a message bus for the subsystems to communicate with each other, and the system does not require a third-party broker or messages to be organized by [**?**]"Topics," which is why Kafka is chosen instead of MQTT. Kafka decouples data streams and systems. The website sends data into Kafka, and the production scheduler queries Kafka for the data. It scales very well and will not require integration with every system.

2) Website: The website subsystem is divided into two parts, where the frontend is responsible for user interaction and display, and the backend which is responsible for request handling, business logic, and communication with other subsystems.

- Backend
  - Programming language: Python Django framework
    - Comprehensibility of the language makes it easier and faster to write understandable code, and easier to comply with the system.
    - It has enough libraries to automate testing, as it can easily conduct testing. Also, it offers a platform that allows automation based on non-browser functionality.
  - Database: Relational Database
- Fronted
  - Programming language:HTML,CSS,JavaScript
    - Runs anywhere, no installation needed.
    - UI updates instantly for all users.

3) Production Scheduler:

- Programming language: Python
- Database: Relational Database(MYSQL)
  - MySQL offers speed and reliability, especially high-speed read operations, where PostgreSQL seems to be more suited for complex queries, data integrity, and consistency. MySQL is better suited for the system, as the system has simple queries and wants speed.
- Non-Relational Database(MongoDB)
  - MongoDB is a document-oriented database. MongoDB stores data in the form of key-value pairs. MongoDB is high-speed, high-availability, and scalability, and also supports JSON formats and allows for ACID transactions.

E. Containerization

This subsection will contain the subsystems that are going to be containerized in the project and how they impact the performance attributes of the system. For containerization, Docker is going to be used.

1) Website: The website has been chosen to be containerized as it has to ensure that the execution environment is the same every time, regardless of the underlying operating system. The website must be scalable since the quantity of users' orders can change over time.

2) Production Scheduler: The Production Scheduler has been chosen to be containerized as it is responsible for coordinating orders, generating production recipes, and scheduling production runs. It ensures the isolation between the scheduling logic and other parts of the system. It improves fault tolerance.

3) Impact: The impact of Docker Compose provides the system with maintainability and scalability. Containerization improves scalability by enabling the deployment of more instances of the Website or Scheduler as system demand rises. It also improves maintainability as it ensures that the service always runs in the same environment and works the same way on different operating systems.

VII. Formal verification and validation

In order to perform a formal verification and validation of the architecure, a model is created to represent the system, based on the Finite-State Machine defined in section **??**.

Timing constraints are identified for the events within the model, based on the requirements. Then, the formal model can be verified to see whether it upholds the constraints and requirements, by simulating it in the model-checker tool UPPAAL [**?**]. The UPPAAL model in this paper was designed to test cover the sequence starting when a customer places an order till it is saved and marked as 'ready' in the system. This will verify the logic of our system, before the architecture is implemented in a prototype. For this purpose, we've created three templates, which we introduce below.

Figure **??** shows a template for a customer. It should be noted that the templates won't show every single possible action that could happen, as the state space grows very quickly the more complex the model is. Testing a large state space isn't feasible, and thus the models are designed only to show the flow of a customer browsing the website and placing an order. The customer template is used only to assist in simulating the flow of the system. Therefore, random invariants have been assigned to the different states. The value 'g' is a global clock, which is reset whenever the customer places an order.

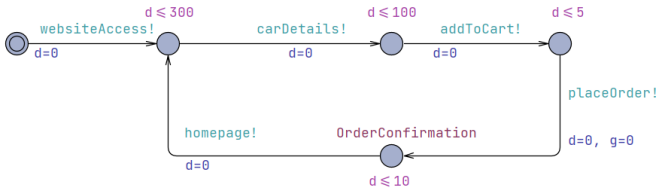

Fig. 7: UPPAAL template for customer - fix size??

Figure **??** and **??** shows the templates for the website and scheduler, respectively.

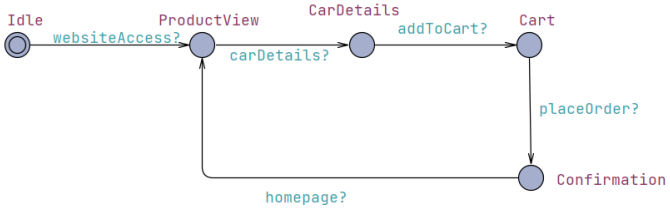

Fig. 8: UPPAAL template for website

The website only changes in response to customer actions. Therefore, any state transition is triggered by a synchronization from the customer template.

The properties to be evaluated, written in the UPPAAL query language, can be seen in table **??**, along with the result from the verification.

ID	Property	Result
UP1	A[] not deadlock	Pass
UP2	E <> (Scheduler.OrderProcessed and $g <= 10$)	Pass
UP3	E <> (Scheduler.OrderProcessed and $g > 10$)	Fail

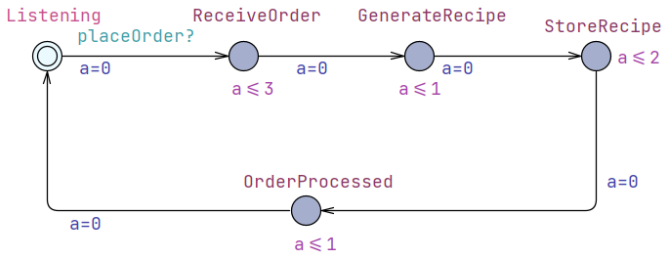

Fig. 9: UPPAAL template for scheduler

UP1 verifies that there is no state in the model where a deadlock occurs. UP2 verifies that there exists some path where the scheduler has reached the 'OrderProcessed' state while at most 10 timeunits have passed (measured using global clock 'g'), since the customer placed an order on the website. Lastly, UP3 verifies the same that U2, except it checks if more than 10 timeunits have passed since the order was placed. Since UP3 fails, this shows that there is no path in our model, where it takes more than 10 timeunits for a placed order to be processed and saved by the scheduler.

VIII. Evaluation

This Section describes the evaluation of the proposed design. Section **??** introduces the design of the experiment to evaluate the system. Section **??** identifies the measurements in the system for the experiment. Section **??** describes the pilot test used to compute the number of replication in the actual evaluation. Section **??** presents the analysis of the results from the experiment.

A. Experiment design

The experiment design will be based off the process presented in Claes Wohlin et al. 2012**. Thus, our goal template will be:

- Analyze our software architecture
- For the purpose of checking whether the requirements are met with respect to the performance QA
- From the point of view of the customer and developer
- In the context of a customer using the system (website and scheduler) to place an order for a customized car.

The scenario we'll be testing is: "From the moment a customer has placed an order on the website, the order must be placed in 'queue' (recipe should be saved in DB, and orderStatus should be 'ready'), within 10 (timeunits**)".

B. Measurements

The time will be measured in (mili)seconds for an action to finish.

C. Pilot test

D. Analysis

IX. Conclusion

Conclusion of the report, discussion and relevant future work.

A. Discussion

B. Future work

CONTRIBUTIONS

Name	Contribution
Anne-Marie	Scheduler, Kafka, Problem and Approach, Related Work, Use case, Design and Analysis Modelling, Formal Verification and Validation