|  |
| --- |
| IUBH |
| Industrial Automation |
| DLMDSINDA01 |

# Learning Objectives

Production systems can be described as discrete event systems where evolution is characterized by the occurrence of events. In the era of Industry 4.0 and highly-flexible manufacturing, there is a need for adequate means for the modeling, analysis, design, and control of flexible production environments.

This course **Industiral Automation** introduces several modeling approaches for the mathematical description of discrete event systems, such as automata, Petri nets, and Markov processes. Each approach is presented in both theory and practice with examples taken from the industry. The approaches are grouped into logic—where only the logic sequence of events determines evolution—and timed—where the time schedule of events also plays an important role. Although simple discrete event systems can be analyzed mathematically, the analysis of complex systems requires the support of computer simulation. The main issues concerning the simulation of discrete event systems will also be addressed in this book.

The final part of this course introduces the concept of supervisory control, which aims at changing the properties of a given system in order to improve specified behaviors and fulfill defined design specifications. Supervisory control is addressed both from a theoretical and from a practical perspective, describing how it can be implemented in a modern industrial environment.

The course ends with a discussion of interesting applications for modeling and design approaches, e.g., in the modeling and analysis of an industrial production unit. Additional conversation on topics like fault diagnosis, decentralized and distributed supervision, optimization, and adaptive supervision provides a contingent connection between classic industrial automation and the recent, (big) data-driven, flexible, Industry 4.0 advanced industrial automation.

# Unit 1 – Introduction to Production Systems

**Study Goals**

On completion of this unit, you will have learned …

… the basic concepts of industrial automation.

… the current and future state of supervisory control.

… the trends in the field of industrial automation and the rise of Industry 4.5.

… the challenges of implementing industrial automation during Industry 4.0.

# 1. Introduction to Production Systems

## 

## Introduction

Industrial automation can be defined as the use of technologies and automatic devices to control machines, systems, and processes. Although it can be argued that automation itself dates back to 3200 B.C. with the invention of the wheel, automation within the manufacturing industry was introduced in 1913 with the Ford Motor Company’s assembly line. This method of automation, where the organization produced a large number of vehicles each year, can be referred to as fixed automation. With this fixed, repetitive automation model, Ford was initially limited to producing only one type of automobile with the Model T before introducing the Model A in 1928.

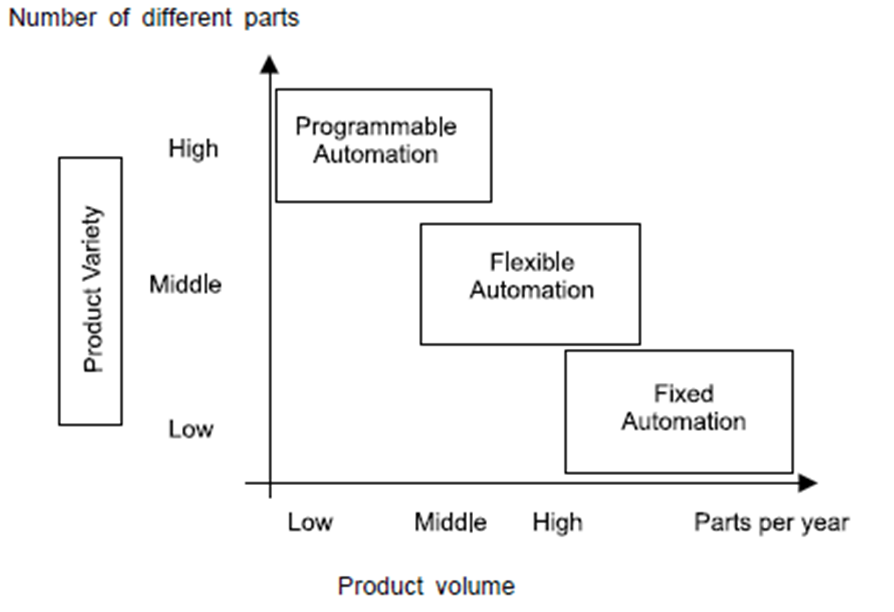
As automated production systems emerged within the manufacturing industry, systems were classified into three distinct types: fixed automation, programmable automation, and flexible automation. In using the previously mentioned Ford production model, many organizations continue to use this method of automation as the fixed automation model is aligned with the organization’s business strategy. It was not until the 1930s that switches, relays, and timers were introduced into the industry, giving rise to industrial automation. The introduction of computational tools and technology allowed for organizations to become more efficient, reduce downtime, and make environments safer through the use of technology (“Automation,” 2020).

With customer demands requiring organizations to create models to address the need for mass customization, leaders of these companies began to integrate machines that could be reconfigured through programmable automation (“Automation,” 2020). The goal of programmable automation was to produce a larger variety of products without the capital expense to purchase additional machines. To minimize the time lost when reprogramming and reconfiguring the organization’s machines and systems while changing over tools and fixtures to produce new lines of products through the programmable automation approach, flexible automation was introduced into manufacturing environments. As an extension to programmable automation, flexible automation enabled organizations to change over configurations to produce new products with no downtime or wait time due to the off-line configuration of the machine. This eliminated the need to reprogram the machine before beginning production on the new product (“Automation,” 2020).

## 1.1 What Are Production Systems?

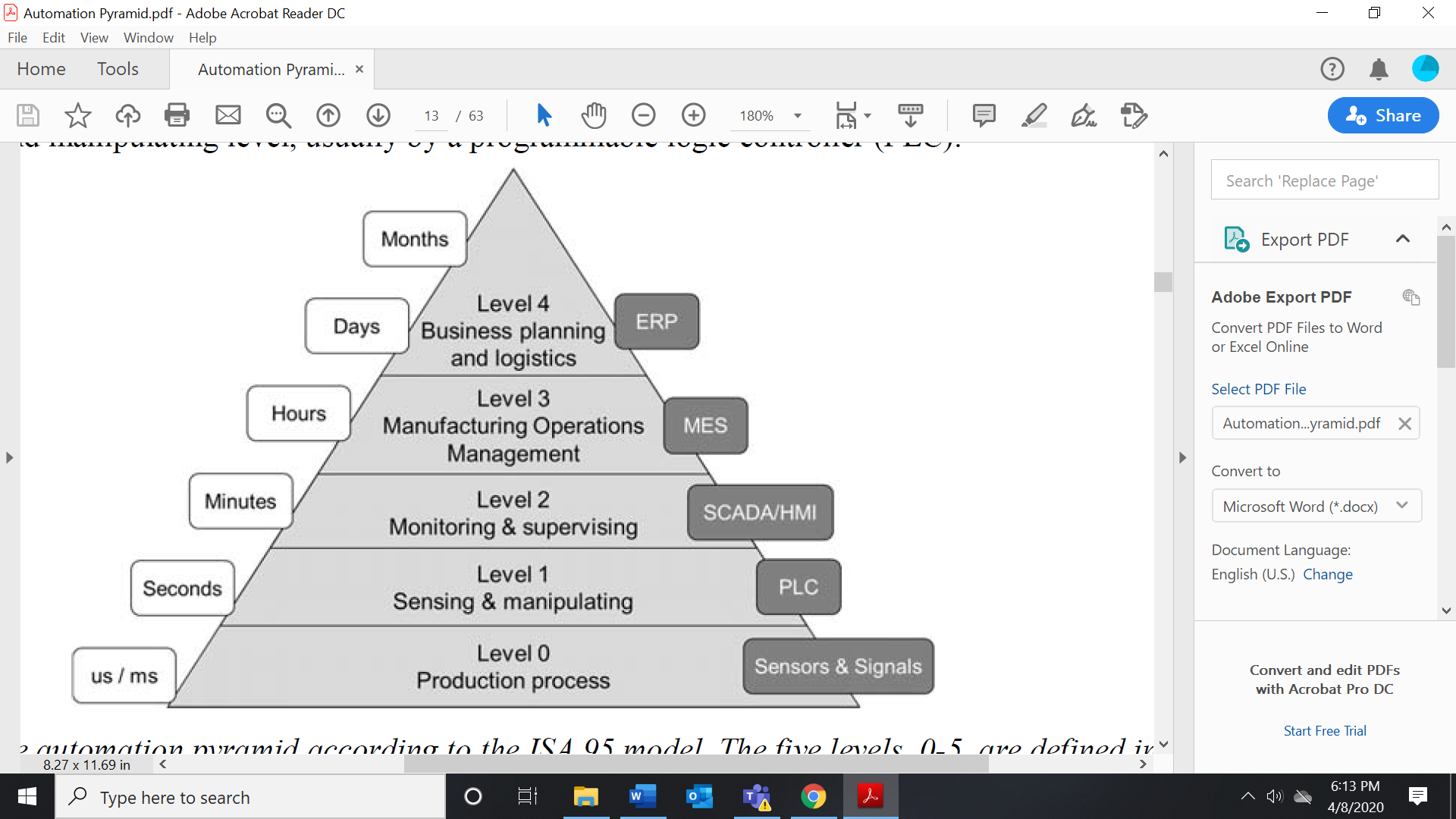
Although group technology integrates many of the tools we will discuss later on, there is no one-size-fits-all approach to the different types of automation systems. The figure below provides a representation of all three production automation types as compared to an organization’s product variety and volume.

Types of Production Automation



Similar to international standards for quality instituted by the International Standards Organization (ISO), the International Society of Automation (ISA) was established in 1945—two years before the founding of the ISO—to develop standards in the area of industrial automation. The standards, referred to as the ISA-95 series of standards (also known as ANSI/ISA-95), were developed by global manufacturers and outline the industry-recognized definitions for automation layers (Hollender, 2010). As organizations continued to integrate devices to operate and monitor the machines within these automation systems and improve the response time of these tools, industry leaders began to separate the families of devices into what is now known throughout the industry as the authomation pyramid (Åkerman, 2018).

The Automation Pyramid



In 2010, many organizations automated their factories and organizations through products and hardware (Åkerman, 2018). Nowadays, the same processes are run by software with the introduction of cloud computing, the Internet of Things (IoT), and the Industrial Internet of Things (IIoT). With the introduction of 5G, developing technologies will allow real-time feedback and response from sensors and signals on the production floor to an organization’s enterprise resource planning (ERP) systems and applications. Although automation tools have evolved and will continue to evolve, the following descriptions provide a high-level overview of the levels and tools within those levels of the automation pyramid.

#### Level 0 – The production process

At the bottom of the pyramid, we begin with the devices that control the production processes—also known as the field-level devices outside of a traditional manufacturing or production environment. The devices used at the production or field level include controllers, sensors, actuators, and motors used to run machinery. These devices ensure the safety of the machinery and its operators, along with assuring that the products are produced within the required specification. In some environments, the data from sensors such as temperature sensors, flow rate sensors, and level sensors do not send information to the manufacturing execution system (MES), requiring manual monitoring on the production floor. As organizations move to integrate sensors and signals to the organization’s ERP applications, this data can be used for sustainability reporting, an area gaining more visibility within the manufacturing industry. Level 0 devices also

* drive the real-time, deterministic communication requirements in some environments;
* measure the process variables and control process outputs; and
* exist in challenging physical environments that drive topology constraints (Cisco Systems Inc., 2020).

#### Level 1 – Sensing and manipulating

Level 1 devices manipulate manufacturing processes through interfaces between Level 0 devices (Raptis et al., 2019). Developed to replace hardwired relay-based controls in discrete environments, PLCs (programmable logic controllers) are used to automate binary processes. To address similar requirements in process-based environments, distributed/decentralized control systems (DCSs) were introduced into the industry. PLCs and DCSs have continued to be one of the most used control devices in the manufacturing industry due to the ability to operate in severe environments, such as those with extreme dust, moisture, or heat. Inputs into PLCs include pushbuttons, proximity switches, pressure sensors, and temperature sensors where outputs are motors, lights, and pumps. Using PLCs and DCSs offers the following advantages (Medida, 2008).

• These devices can record and store large amounts of data.

• The data can be displayed in any way the user requires.

• Thousands of sensors over a wide area can be connected to the system.

• The operator can incorporate real data simulations into a system.

#### Level 2 – Monitoring and supervising

Supervisory control and data acquisition (SCADA) systems are used to control devices and sensors. In traditional environments, SCADA systems integrate data acquisition systems with data transmission systems with human machine interface (HMI) software to provide a centralized monitoring and control system for numerous process inputs and outputs. HMIs are used to communicate and display the status of a machine in a human-readable format to the individual operating the machine (Raptis et al., 2019). Common HMIs within production environments are screens to view and adjust set point values, temperatures, and flow rates, among others.

#### Level 3 – Manufacturing operations management

At the next level of the pyramid, plant supervisors and line managers monitor and plan production through an MES. Depending on the solution deployed, an MES allows organizations to make real-time decisions within the organization’s production based on information received from SCADA systems and PLCs (Raptis et al., 2019). In the MES, organizations process data coming from devices on the shop floor to calculate performance indicators, such as the overall equipment effectiveness (OEE), and to regulate process control systems, among other processes. Additional benefits to MESs include

* the elimination of waste,
* the reduction of paperwork errors due to the creation of a paperless environment,
* the ability to systematically standardize operational processes throughout an organization’s supply chain network,
* the ability to accelerate the root cause diagnosis and resolution of product and process issues, and
* increased control over an organization’s process.

#### Level 4 – Business planning and logistics

At the top of the pyramid, we have the business planning and logistics functions. This level is where the information from the signals and sensors of the machines on the plant floor are sent to managers, directors, and leaders of an organization to make tactical and strategic decisions in an application known as an ERP system. ERP systems are integrated, customized, and packaged software-based systems that handle the majority of system requirements in all functional areas of a business such as finance, human resources, manufacturing, sales, and marketing. In addition to using ERP systems as a tool to make day-to-day business decisions, these systems can also be used as a tool to improve knowledge sharing within the organization (Goldston, 2020). With ERP applications, organizations will enable departments and facilities to share knowledge and collaborate instead of operating out of disparate systems.

### Industry 4.5 – Integrated Intelligent Automation

In identifying the manufacturing industry as a laggard in technology and innovation as compared to other industries such as healthcare, in 2011, the German government created a project called “*Industrie 4.0*,”also known as Industry 4.0, to promote the integration of technology within manufacturing (Giallanza et al., 2020). In an effort to move out of the traditional automation age of Industry 3.0, it was the vision of the German government to create what some called a “smart factory” that would include cyber-physical systems, cloud computing, and the IIoT, resulting in interoperability, the decentralization of information, real-time data collection, and increased flexibility.

In 2010, many manufacturing organizations automated their factories and field operations through products and hardware. As the industry entered 2020, the same processes were controlled by software (Durakbasa & Gençyılmaz, 2021). As leaders of organizations identified use cases for technologies such as artificial intelligence (AI), machine learning (ML), and edge computing for industrial application, a shift began from hardware applications to software applications, hence the terms IIoT or smart factories (Durakbasa & Gençyılmaz, 2021). Although the delivery method changed from hardware to software, the functionality and logic for each step of the automation pyramid are still required. Organizations continue to make this transition at level 4 of the automation pyramid where companies are moving their ERP systems from on-premise applications to the cloud. In doing this, leaders have reduced the overhead and maintenance costs incurred by housing servers within their organization to a software-as-a-service (SaaS) model where the company can pay a subscription fee for its vendor to host, maintain, and upgrade the organization’s systems. Although organizations have implemented aspects of the German government’s vision, a large majority of manufacturers continue to use the devices and tools of Industry 3.0 effectively (Durakbasa & Gençyılmaz, 2021).

As we close in on the first decade of Industry 4.0, although small steps have been made to align with the German government’s vision, the next decade may introduce the next phase of Industry 4.0 with Industry 4.5 (Raptis et al., 2019). With the increasing inclusion of AI, ML, blockchain, along with other technologies, within various industries, the manufacturing industry will see an emergence of integrated intelligent automation (IIA).

When performing a search on the academic research on intelligent industrial automation, the search returned 759 results. When performing a similar search on integrated intelligent automation, the search returned only twelve results. Given that integration is a common challenge researchers have, the integration aspect is one that will be a primary focus of the literature when discussing integrated intelligent automation. With current research identifying cybersecurity as a primary concern with Industry 4.0 in various industries, as well as with the introduction of cyber-physical systems (CPSs) and cyber-physical production systems (CPPSs), IIA may be an extension to CPSs or CPPSs in Industry 4.5 (Elhabashya et al., 2019; Lu et al., 2016). With IIA, Industry 4.5 will introduce AI, business process automation (BPA), robotic process automation (RPA), and the integration of devices identified in the automation pyramid to create the smart factory envisioned at the onset of Industry 4.0.

#### Robotic process automation

RPA will be integrated within manufacturing environments to perform routine, repetitive tasks, enabling employees to perform the day-to-day decision-making tasks required within the organization. Global research and advisory firm, Gartner, noted that by 2022, 65 percent of organizations will introduce AI, ML, and natural language processing (NLP) into RPA tools (Ray et al., 2019). RPA will allow organizations to reduce errors and increase efficiency within their production operations. From the automation of validating accurate bills of material (BOMs) and routings for production planning to the production of the parts, the inclusion of RPA within manufacturing environments will lead to reduced costs, increased profitability, and opportunities for individuals within an organization’s respective communities (Ray et al., 2019).

Outside of the production department, RPA will allow other departments within organizations to become more efficient and reduce errors (Chakraborti et al., 2020). For example, within the accounting department, accounts payable and accounts receivable invoices can be automated for review, approval, and payment once fields on the invoices are mapped to the defined system for approval. In the customer service department, RPA could effectively and efficiently review hundreds or thousands of sales order lines to ensure the accuracy of products, descriptions, prices, and discounts. This process will reduce internal processing lead times and errors, while at the same time increasing customer satisfaction by quickly identifying discrepancies and outliers.

**Business process automation**

Similar to RPA, the goal of implementing BPA is to increase the accuracy, efficiency, and compliance of business processes (Chakraborti et al., 2020). BPA will be integrated with ERP systems, MES systems, sensors, and applications to submit alerts and notifications from all parts of the world to individuals within an organization. Additionally, for organizations operating in a global environment, hardware and software can also be monitored and maintained from anywhere around the world.

In reviewing BPA and RPA, it is important to remember that, although both will streamline processes and reduce errors, RPA focuses on individual tasks while BPA streamlines an entire process. In both offerings, organizations can implement a supervised, unsupervised, or hybrid automation strategy to further increase efficiency within the organization. With an attended automation strategy, production operators can work side by side with robots and technology to complete tasks at greater speed and accuracy (Chakraborti et al., 2020). In using an unattended strategy, as mentioned in the RPA section, organizations can automate repetitive tasks, such as inventory picking and putaway on the production and warehouse floor. With a hybrid automation strategy, organizations can use a combination of both approaches to perform activities. For example, in an automotive environment, line workers could inspect and place a fastener on a component where a robot could apply the correct torque based on specifications.

## 1.2 Industrial Supervision and Control

SCADA systems have been a part of industrial automation since the onset. First utilized by panels of meters, SCADA systems continue to be used to perform supervision and control for manufacturing plants and facilities (Medida, 2008). Providing real-time monitoring of remote sensors, systems, and actuators across an organization’s supply chain network, as well as in the field, SCADA systems use remote terminal units (RTUs), which are small computerized units deployed in the field at specific sites and locations. RTUs serve as collection points for gathering data from sensors and deliver commands to control relays to be sent back to the centralized server within the organization.

There are typically five tasks in any SCADA system and each of these tasks performs its own separate processing (Strauss, 2003).

• Input/output task: This program is the interface between the control and monitoring system and the plant floor.

• Alarm task: This task manages all alarms by detecting digital alarm points and comparing the values of analog alarm points to alarm thresholds.

**•** Trends task: This task collects data to be monitored over time.

• Reports task: Reports are produced from plant data. These reports can be periodic, event-triggered, or activated by the operator.

**•** Display task: This task manages all data to be monitored by the operator and all control actions requested by the operator.

Within traditional environments, SCADA systems are limited by the organization’s centralized network topology, requiring additional hardware to allow for scalability. With IIoT, organizations are replacing RTUs with cloud-based SCADA systems through the use of offerings such as Amazon Web Services (AWS). In one example, comparing this solution to software a SaaS, Langmann and Stiller (2019) referred to this offering as smart industrial control services (SICS). In their study, Dugyala et al. (2019) outline the following benefits in making this transition.

* Security: New, advanced encryption approaches allow for a multi-tiered security model.
* Scalability: Moving SCADA capabilities to the cloud will not lead to latency issues due to the growth and expansion experienced by organizations utilizing on-premise solutions.
* Cost: In using a subscription-based model as used with many cloud providers, the cost of maintaining applications in the cloud will save organizations the additional capital required to maintain applications on-premise and in the field.

## Self-Check Questions

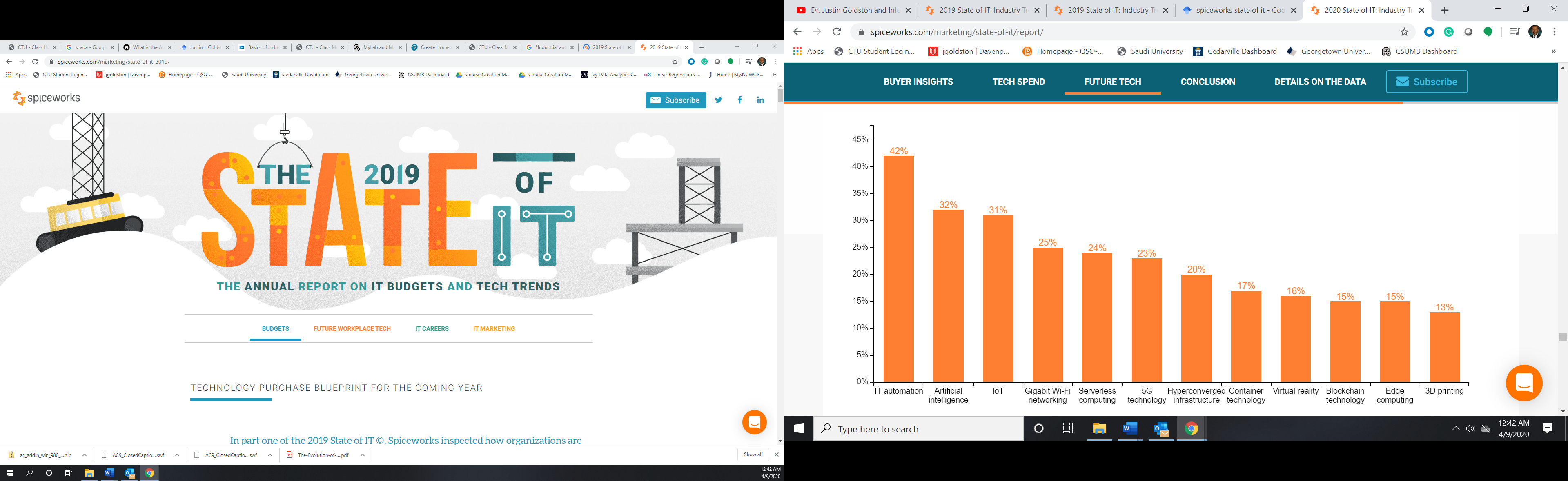
1. What is the role of the alarm task of a SCADA?

*The task manages the alarms by detecting digital alarm points and comparing the values of analog alarm points to alarm thresholds.*

## 1.3 Trends

In a survey by Spiceworks, Inc., the figure below outlines the technological trends that will make the biggest impact in 2020. In analyzing the responses from over 1,000 IT professionals from small, medium, and large enterprises across the United States and Europe, IT automation topped the list, followed by AI and IoT (Spiceworks Inc., 2020). Although a number of different emerging technologies were presented in this survey, in pursuit of an integrated, intelligent automation environment, we will see how many of these tools are integrated through hyperconvergence—another offering covered in the survey.

Technology Trends Expected to Have the Biggest Impact on Businesses in Total



**IT automation**

As computers were introduced in the 1960s, organizations began to develop applications to track inventory, assist in ordering materials, and produce finished goods. In a concept identified as inventory control, firms took the first step in systematically running the operational side of their organization (Goldston, 2020). In the 1970s, materials requirements planning (MRP) applications were introduced to enable organizations to purchase, forecast, and schedule production, spawning the founding firms of the industry such as SAP and J. D. Edwards. With the number of organizations creating additional requirements to reduce their overhead costs, J. D. Edwards enhanced their MRP applications to include closed-loop scheduling, enhanced shop floor reporting, and forward scheduling known as MRP-II (Goldston, 2020). As organizational leaders began to revert to technology to assist in daily operational decision-making, the primary ERP vendors—SAP, IBM, J. D. Edwards, Baan, PeopleSoft, and Oracle—were established by the end of the 1980s (Razzhivina et al., 2015). With enterprise applications enabling decision-makers to provide better visibility of their inventory and production levels, organizations also looked to these applications to set themselves apart from their competition.

As previously mentioned, many organizations within the industrial sector continue to operate with an Industry 3.0 mindset, where although these environments have certain levels of automation, devices operate through centralized logic processors and information technology. This centralization leads to the requirement of human input and monitoring, leading to high overhead costs, wasteful processes, and activities, and the risk of defects for manufactured products (Durakbasa & Gençyılmaz, 2021). With the inclusion of offerings such as cloud and edge computing, decentralized architectures will allow organizations to take steps toward Industry 4.0 with the goal to create an IIA-focused environment.

**Artificial intelligence**

As we have previously discussed, AI is changing the way organizations operate and make tactical and strategic decisions. Although the term “artificial intelligence” dates back to the 1950s, with the rise of AI within the industry, AI became a buzzword at the end of the second decade of the 21st century. When organizations continued to find viable use cases for the technology, AI became a must-have to remain competitive within their respective markets. Within manufacturing environments, with the ability to increase efficiency and reduce costs, manufacturers also found AI to be a competitive advantage to pass cost savings to customers.

In a series of use cases applicable to manufacturing environments, Ben-Assa (2019) highlighted five examples of where organizations can introduce IIA and AI into production environments.

1. Misplaced parts, tools, and fixtures: In many discrete manufacturing environments, organizations use Lean 5S—sort, set in order, shine, standardize, and sustain—to keep track of tools and fixtures. Given it may not be feasible to place a sensor or RFID device on every tool or fixture, AI and machine learning can identify locations, lines, and work centers where tools and fixtures are most commonly used and alert users when deviations occur.
2. Improved quality through AI: Expanding on the previously outlined Lean 5S approach, organizations use a variety of additional lean tools to identify root causes and errors in products and processes. With the help of AI, data analysts within organizations can perform root cause analysis using AI-based algorithms. As AI continues to analyze and resolve issues, the tool creates a knowledge base, resulting in predictive analytics to allow AI to alert users of potential machine failures and product defects before they happen.
3. Identifying, predicting, and preventing bottlenecks: Traditionally, manufacturing organizations depended on MES systems to estimate the time to produce products. To calculate production lead times, MES systems retrieved data from a product’s bill of routing, which was often inaccurate, leading to delayed orders. In an industrial environment, **finite capacity planning** models are rarely implemented. This decision results in late orders due to unavailable tools and fixtures, expired and unavailable products, or unavailable machines since ERP and MES systems do not “see” this real-time activity. Through the real-time, continuous data collected from all devices and systems of an organization, AI can alert users of delayed orders, update bills of routings, and predict maintenance of machines based on the historical data.

**Finite capacity planning**

A method of planning production that assumes that the available capacity is a defined constant.

1. Optimized planning and production:Referred to as cut planning or nesting in production environments, some organizations struggle with the exercise to identify the most cost-effective way to cut or stamp the most parts out of a sheet of raw material such as steel or carbon microfiber. Although software programs have come a long way to remedy this issue, AI can assist in this activity by identifying patterns of the highest-yield runs based on previous jobs or production orders.

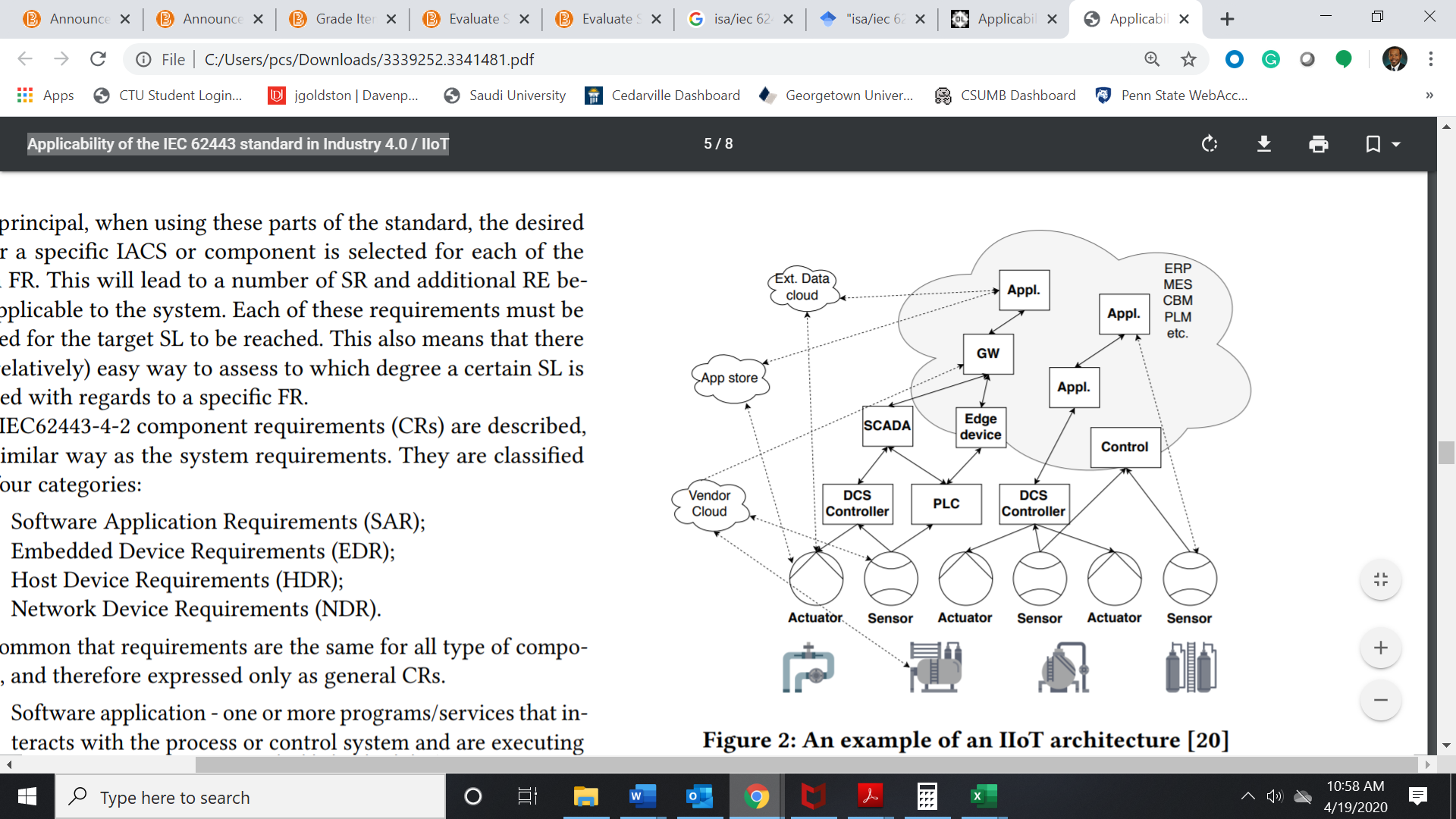
With AI providing organizations with unprecedented insights and visibility into information to make tactical and strategic decisions, one of the most important, often overlooked inputs are the data. As the AI solution collects more data from the organization’s devices, AI will be able to identify patterns and make more accurate predictions as the application continues to “learn.”

**The Internet of Things**

In a broad definition, IoT can be defined as interconnected devices that allow for increased efficiency within an individual’s everyday life (Firouzi et al., 2020). With IIoT (Industrial Internet of Things), we bring that efficiency into organizations and use IIoT as a foundation for IA and AI. As the capabilities for IA and AI continue to develop, the vision of smart factories may shift to autonomous factories in the near future. This idea dates back to the early 1980s when General Motors (GM) chairman, Roger B. Smith, shared his vision of “lights-out manufacturing” with the U.S.-based newspaper The New York Times (Holusha, 1982). Although this failed experiment resulted in robots painting themselves instead of cars, the company that sold Smith the robots—Japan’s Fanuc Ltd.—performed a lessons-learned assessment of GM’s experience and perfected the process. In the early 2000s, Fanuc implemented an unsupervised RPA process where robots built other robots at a rate of 50 per 24-hour period shift for 30 days at a time (Null & Caulfield, as cited in Olsen & Tomlin, 2020).

With nearly 20 years of application of RPA processes, coupled with the inclusion of IIoT and AI, the reality of lights-out manufacturing is on the horizon. In taking steps to an autonomous industrial environment, organizations have begun to enable devices, sensors, and systems with IoT. In providing real-time data collection of feeds such as temperature, humidity, and other environmental conditions, sensors and devices can alert users of the current status of a product or machine, as well as deviations and outliers through the means of ERP applications, tablets, smartphones, or smartwatches through application services created by the vendors of the devices and sensors. With the various capabilities emerging within the industry, the figure below outlines a high-level model of how an IIA architecture may be structured.

Future-State Integrated Industrial Automation Systems



### Additional Trending Topics

As broader topics continue to be introduced as it pertains to IIA, the integration of tools such as AI and ML into existing tools such as ERP applications will provide additional capabilities for organizations as they enter into Industry 4.5. Below are additional options that could be considered when implementing an IIA strategy.

**Predictive maintenance**

With the evolution of ERP applications, organizations began to track the preventative maintenance of field- and plant-level machines, devices, and sensors within ERP applications with a goal to reduce downtime and overhead costs. Although ERP providers continue to develop preventative maintenance capabilities within the organization’s offers, it has been found that preventative maintenance is only 15% successful, and in some environments, results in random failure rates as high as 85% (Goldston, 2019a).

In one study, Amonkar (2019) found that, by scheduling maintenance early, an organization may be wasting machine life that is still usable, thereby adding to overhead costs. The author also provided an example of a plant with 5,000 measuring points and over 1,000 positioners, 500 temperature sensors, 150 flowmeters (Amonkar, 2019). How does such an organization know which devices take priority? Through the use of sensor data to identify service needs, an algorithm-based pre-selection model for each device can be created with AI. Utilizing trends and histograms, the data can be collected and monitored by ML to create a predictive maintenance model.

By implementing a predictive maintenance model in a production environment, in comparison to the traditional predictive maintenance approach, organizations such as Lockheed Martin were able to detect anomalies in data more efficiently, prevent downtime, and predict failures with 95% accuracy. In another case study, with the introduction of predictive maintenance, Anheuser-Busch InBev was able to identify when filter degradation impacted clarity of beer and increase filter run length by 40%-50% (Meena, 2019).

**Blockchain**

Although many individuals have heard of Bitcoin, the large majority are unfamiliar with the technology it is built upon—blockchain. An enterprise blockchain is a distributed ledger that maintains a list of every transaction across a network of thousands of computers in some cases. Because blockchain transactions are immutable and cannot be changed, coupled with the fact that the blockchain becomes more secure with time, these attributes make blockchain offerings appealing to manufacturing organizations operating in a digital environment. In one of the most notable case studies on blockchain and supply chain management, in an exercise that would normally take them 6 ½ days, Walmart was able to recall one batch of mangoes in 2.2 seconds using blockchain technology they developed in collaboration with IBM.

As organizations began integrating blockchain applications across supply chain networks around 2019, companies in industrial sectors—such as aerospace and defense—began securely transferring documents and serial numbers throughout an airplane or helicopter’s bill of material. Also, as organizations developed smart contracts (self-performing contracts based on agreed rules) to automatically procure products once the reorder point is reached, a similar approach could be implemented in an IIA environment. In what could be referred to as a BPA approach, organizations could develop agreements with maintenance, repair, and operations (MRO) vendors to create smart contracts to automatically issue orders for supplies such as oil and lubricants, as well as repair requests for devices, sensors, machines, and robots.

## 1.4 Challenges

Although legacy devices such as PLCs and DCSs can store large amounts of data, the data alone do not provide the level of information needed to make tactical and strategic decisions. With cloud computing and IoT, IT professionals can now receive real-time analytics from production and field-level devices to dashboards within the organization’s ERP applications or through tools previously mentioned, such as BPA. As innovators within the industrial sector reflect on their respective organization’s industrial automation strategies, researchers have reached a consensus as to four common challenges: integration, scalability, interoperability, and security (Åkerman, 2018; Carvajal Soto et al., 2019; Hofer, 2018; Trunzer et al., 2019). With over 120 serial communications standards and upwards of 30 Ethernet-based protocols offered to manufacturers, the combinations of industrial communications protocols that are “supported” by various vendors complicate the challenges and complexity (Gurrapu, 2017).

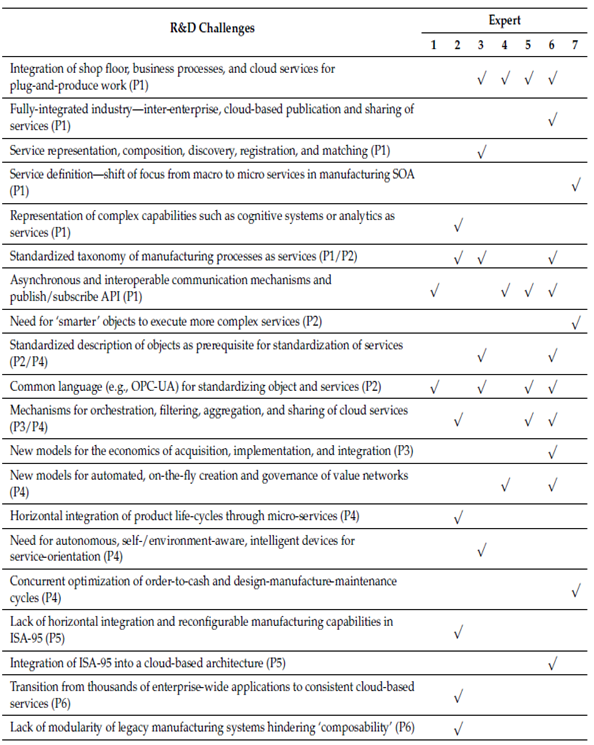
As organizations must be flexible within an ever-changing environment with increased competition and customer demands, technology will not only create a competitive advantage, but it will also harness growth and creativity throughout the company among the organization’s workforce. To successfully implement these digital transformation initiatives, leaders should be knowledgeable of the challenges identified above and develop strategies to counter those challenges.

**Integration**

As many manufacturing organizations, namely the automotive, oil and gas, pipeline, and water treatment industries, continue to use legacy control systems, proprietary DCS, SCADA, and PLC devices cannot be integrated with other devices due to outdated technology (Dickman, 2009). The emergence of CPS introduced the intersection between virtual and embedded technology with systems, machines, and hardware. Although this is yet another step in creating an integrated industrial environment, the heterogeneous solutions from various software vendors have been found to create issues and latency in response time between applications (Gunes et al., 2014; Hofer, 2018).

The table below outlines a survey of challenges encountered by industry experts when implementing technologies within an industrial environment.

A Survey of Integration Challenges



In reviewing the results of the study, it can be concluded that a number of considerations should be taken into account when planning for a digital transformation. It is also important to note that two of the most cited issues were the integration of applications and devices and the standardization of M2M (machine-to-machine) communication protocols. In using an industrial standard Open Platform Communications – Unified Architecture (OPC-UA), organizations can alleviate these challenges. Developed by a consortium of engineers and end-users, the OPC Foundation established standards to provide scalability and extensibility of feature-rich open platform architecture (Moghaddam et al., 2018).

In outlining the goals of the standards for OPC-UA, Robles et al. (2015) highlighted the following attributes of OPC-UA.

• Functional equivalence: all OPC Classic (OPC-Data Access (**OPC-DA**)) specifications being mapped to UA

**OPC-DA**

This is a specification that defines how real-time data can be transferred between data sources without the knowledge of a system or device’s native protocol within the network.

• Platform independence: the shift from an embedded micro-controller to cloud-based infrastructure

• Secure: the advanced encryption, authentication, and auditing

• Extensible: the ability to add new features without affecting existing applications

• Comprehensive information modeling: modeling for defining complex information

By deploying an OPC-UA approach, organizations can implement standardized protocols to create a more integrated industrial environment.

**Scalability**

Although CPSs provide a level of industrial automation, scalability is still a concern. As organizations continue to add integrated and intelligent automated sensors, stations, and production lines, the amount of data collected throughout an organization’s enterprise will increase. In addition to the amount of data collected, the frequency at which the data are pushed to users will also shift to a real-time data collection model (Carvajal Soto et al., 2019). As intelligent devices will integrate with applications and systems throughout an organization’s global supply chain network, technology providers are collaborating with organizations and researchers to incorporate edge computing and cloud computing in industrial environments with 5G technology to achieve the real-time data exchange required to effectively monitor and control IIA devices (Cheng et al., 2018).

**Interoperability**

Since organizations continue to convert legacy systems to integrated applications to this day, interoperability will continue to be a topic of concern. Due to the coexistence of new and legacy systems, technical professionals create application programming interfaces (APIs) to manage the flow of information between disparate systems. As an interface that pulls data from unrelated devices, APIs have operated as the “middleman” between applications in various industries since 2000. The manufacturing interoperability program at The National Institute of Standards and Technology (NIST) in the United States outlined a number of factors that impact interoperability within a production environment (Zeid et al., 2019), which include

* the transfer of data between systems being dissimilar,
* the transfer of data between software being made by the same vendor but having different versions on the systems,
* the compatibility—or lack thereof—between different versions of software,
* the misinterpretation of the terminology used or the lack of understanding of the terminology used for the exchange of data or information,
* the use of non-standardized documentation on which the exchange of data is processed or formatted, and
* the lack of testing of those applications that are deemed conformant, due to the lack of means to do so between systems.

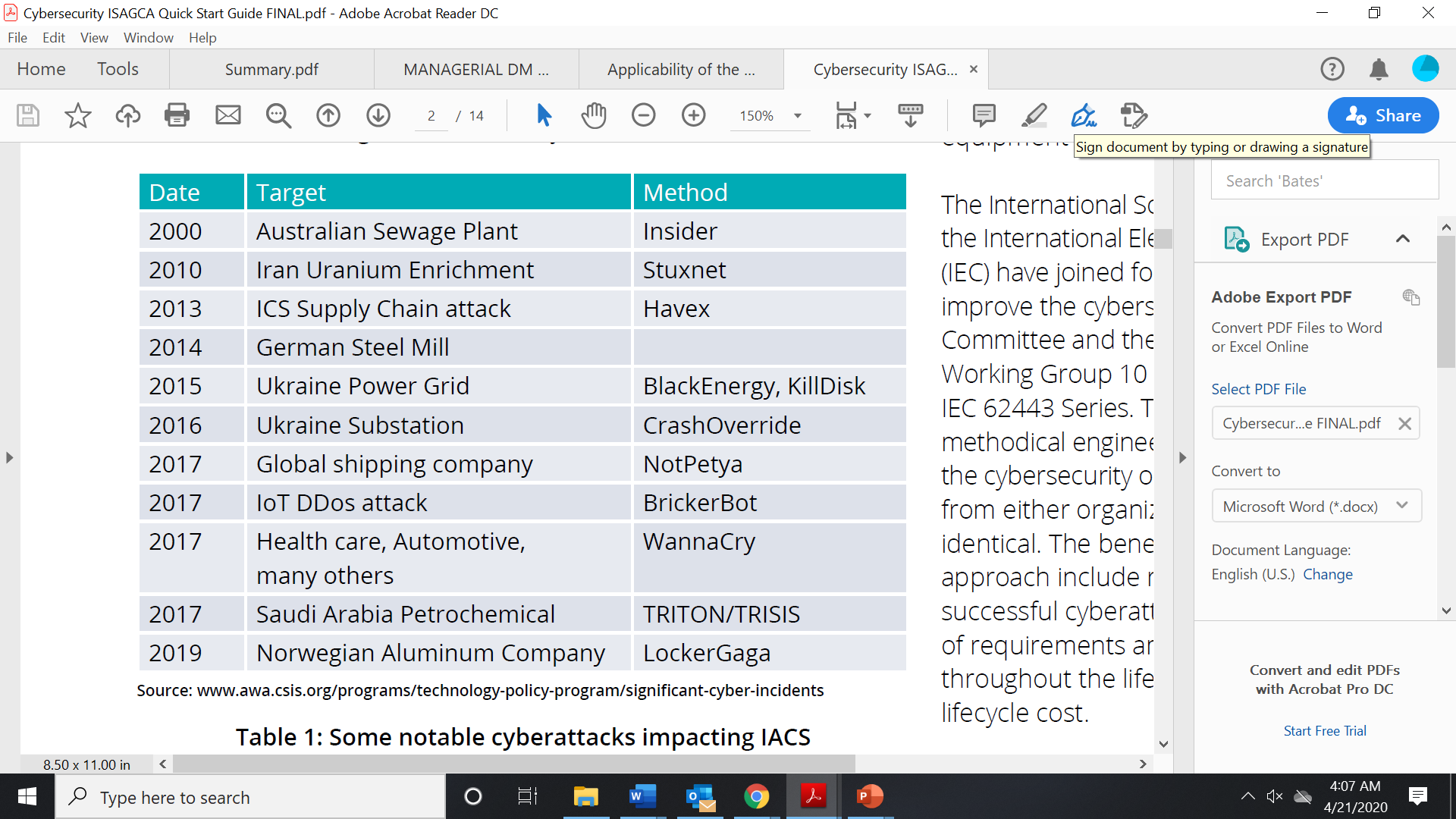
An additional challenge to interoperability could be posed by inconsistent data formats between applications, which could be alleviated by ensuring correct data mapping between applications.

**Security**

As technological change will continue to be researched, it is more important to explore the cyber threats of new technologies and to create mitigation strategies to counter disruptions with the introduction of new, innovative solutions. Leaders of organizations have noted they have delayed lower-level IIoT capabilities due to cybersecurity concerns. In one simulation, IIoT cybersecurity company Trend Micro and Italian technical university Politecnico di Milano (POLIMI) identified the vulnerabilities of over 83,000 industrial robots due to poor authentication, cryptography, or outdated software (Crelier, 2019).

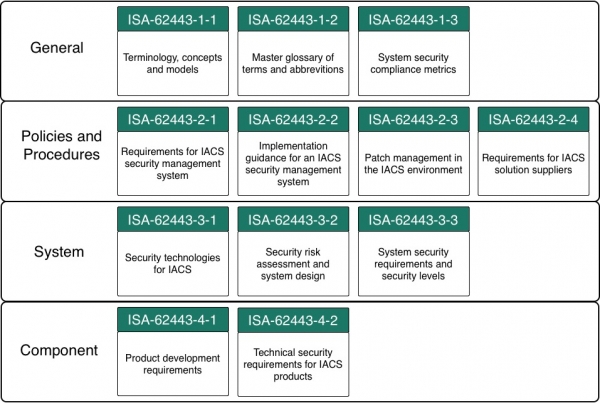
Hackers have also noted that a SCADA system that is running a Microsoft operating system can take less than two minutes to penetrate. With only five to six major DCS and SCADA systems used globally and with millions of engineers trained on the use of these devices, the ability to hack industrial systems becomes alarming (Dickman, 2019). In 2011, an Iranian nuclear plant’s systems were exposed to malware through USB ports, routers, and SCADA systems. In another example, due to a poorly secured Web Gate web service, hackers were able to remotely freeze HMI panels and disconnect them from the SCADA network at Schneider Electric in 2016 (Crelier, 2019). As the world becomes more dependent on technology, organizations and individuals will become more vulnerable to security threats. The table below highlights notable security breaches during the first two decades of the 21st century.

Notable Cyberattacks Impacting IACS



In 2019, ISA announced the creation of the ISA Global Cybersecurity Alliance to promote the ISA/IEC 62443 series of standards, the world’s only consensus-based cybersecurity standard for industrial automation and control systems (IACSs) (ISA Global Security Alliance, 2020). The figure below depicts the standards outlined by the ISA Global Security Alliance.

The Structure of IEC 62443



Beginning at the top, the general layer of the ISA/IEC 62443 series of standards provides an overview of the terminologies and models, along with the metrics that can be used to measure security life cycles of IACSs. The policies and procedures layer outlines the specifications for IACS security with the objective to continuously improve the process through lessons-learned from ongoing risk evaluation and reviewing previous malicious attempts on IACSs. Using the automation pyramid as a reference, the system layer of the ISA/IEC 62443 standard addresses the MES and ERP levels of the automation pyramid. At this layer, a risk assessment is performed on the production-level and enterprise-level applications and vulnerabilities are addressed. The fourth layer—the component layer—refers to the production, sensor, and monitoring levels of the automation pyramid. At this layer, the ISA/IEC 62443 standard reviews the life cycle requirements for IACSs and identifies the technical security requirements for these components.

Given that federal governments are acknowledging the importance of AI’s impact on cybersecurity, and as manufacturing organizations move to more intelligent, autonomous environments, leaders of these organizations should educate their workforce on cybersecurity vulnerabilities. In March of 2020, the National Science and Technology Council published a report stating that, with the rapid advances of technology, the integration of AI, ML, and cybersecurity could be implemented to mitigate the risk of all cyberattacks, not just those on federal governments (National Science and Technology Council, 2020). With these findings, additional research will be required to understand how organizations within the industrial sector can use AI to protect AI-powered systems, enhance security efforts, and to anticipate the use of AI by attackers (Gartner, 2020).

**Skilled labor shortages**

As the technical challenges to the implementation of these innovative technologies have been outlined, the key to maintaining the effective use of these tools will be through employee adoption, empowerment, and job enrichment (Goldston, 2019b). With 40% of skilled manufacturing workers projected to retire by 2025, this figure may increase as employees believe that technology and automation will replace jobs (Lu et al., 2016). Transformational leaders can inspire, encourage, empower, and influence employees to work toward the common objective of a successful digital transformation. Through transformational leadership, leaders of organizations can show how all employees within their workforce will have a role in the organization’s digital transformation.

With the growing number of employees born in the digital age entering the industry, leaders can also incorporate reverse mentoring to allow tenured workers to learn these technologies from the technologically savvy employees. Thus, new hires can learn the processes within the operation and the mentor and mentee can use additional tools such as virtual reality (VR) and augmented reality (AR) to create training videos and visuals to develop future-state processes. Through the use of VR and AR, tools such as Google Glass can be used to check temperatures and sensors, while also walking employees through preventative maintenance and repair tasks created by tenured employees, similar to a virtual product or repair manual. Through the approaches of including technology into training, the collaboration among individuals will create a cultural shift to embrace technological tools through the opportunities to enrich current job functions.

Similar to Google Glass, the concept of digital twins is also emerging as early adopters are implementing this emerging technology in their business environments. Introduced in the automotive industry to reduce costs and time to market, creating a digital replica of a physical item assists in training new personnel and creating simulations for a variety of different use cases. Additionally, with the use of digital twins, organizations can create a digital representation of all aspects of the production process to identify bottlenecks and inefficiencies within the operation before new or modified processes are implemented. As an extension to predictive maintenance, in creating a digital version of each machine on the production floor, organizations would be able to identify trends and anomalies to predict failures.

**Implementation strategies**

Because of the complexity of system integration within industrial environments, the implementation and assimilation process are always associated with high risk (Goldston, 2019c). To analyze an organization’s readiness for an enterprise-wide digital transformation, leaders should understand the company’s current state maturity. As consulting organizations provide maturity assessments for transformations such as ERP implementations, current research indicates similar models have been developed for an organization’s Industry 4.0 readiness. In a multi-methodological development approach, Schumacher et al. (2016) proposed a model of 62 maturity attributes grouped into nine company dimensions. The table below provides an outline of the dimensions with a high-level definition of each of them.

Dimensions of the Industry 4.0 Maturity Model

|  |  |
| --- | --- |
| Dimension | Definition |
| Strategy | Implementation of an Industry 4.0 roadmap, available resources for realization, adaption of business models, … |
| Leadership | Leadership buy-in, management competences and methods, existence of central coordination for Industry 4.0, … |
| Customers | Utilization of customer data, digitalization of sales/services, customer’s digital media competence, … |
| Products | Individualization of products, digitalization of products, product integration into other systems, … |
| Operations | Decentralization of processes, modeling and simulation, interdisciplinary, interdepartmental collaboration, … |
| Culture  People  Governance  Technology | Knowledge sharing, open-innovation and cross company collaboration, value of Information and Communication Technology (ICT) in company, …  ICT competences of employees, openness of employees to new technology, autonomy of employees, …  Labor regulations for Industry 4.0, suitability of technological standards, protection of intellectual property, …  Existence of modern ICT, Utilization of mobile devices, Utilization of machine-to-machine communication, … |

In reviewing the dimensions, it is important to note that, although the maturity assessment is performed to assess an organization’s readiness for an organizational change from a technological perspective, people and process-related factors are more critical to the success of this transition as compared to an organization’s technical capabilities.

|  |
| --- |
| Summary |
| In reviewing the topics discussed in previous sections, organizations have realized the importance of integrating technology into daily processes and strategic planning. Although the devices, systems, and applications have been around for decades, the goal to achieve increased efficiency, reduced costs, and increased quality have made leaders of organizations advance their focus and capital on industrial automation. Due to the fact that a limited number of organizations around the world have integrated and implemented the technologies mentioned in this unit, subsequent units of this text will outline the foundations and the current state of industrial automation, as well as how the future may look as we move to Industry 4.5. |

# Unit 2 – Automata

**Study Goals**

On completion of this unit, you will have learned …

… how formal languages and formal grammars are defined.

… how formal grammars can be classified by the Chomsky hierarchy.

… the difference between deterministic and nondeterministic finite automata.

… two composition techniques to form a new automaton from two existing automata.

# 2. Automata

## Introduction

The theory of automata deals with models of machines that can be used to solve certain sets of problems. It is directly connected with the theory of formal languages and formal grammars because different types of languages can be recognized by different kinds of automata (Salomaa, 2014). This unit will offer an introduction to the underlying theory and the interrelation of different models, it will define the concepts of formal languages and grammars, and show how they can be classified by means of the Chomsky hierarchy. In addition, it will also give you deeper insight into deterministic and nondeterministic finite automata and illustrate how the latter can be transformed into the former. The unit will close by showing some properties of finite automata and how multiple automata can be composed to form a new one.

## Preliminaries

To understand the functionality and purpose of automata we first need to have a look at the basics and mathematical preliminaries. The topic of automata is part of the theory of formal languages, which is an important concept for the mathematical description and analysis of strings. A formal language differs from natural languages in that it is an abstract language that is intended to be used for a mathematical purpose and not for communication. The definition of a formal language is found below (Révész, 1991).

An alphabet Σ is a set of symbols.

A sequence of characters w ∈ Σ\* is called a word (over Σ).

A set of words L⊆ Σ\* is called a language (over Σ).

A grammar of a language L describes all the rules for the generation of words of L.

**Kleene star**

This operator is named after the American mathematician and logician Stephen Cole Kleene.

The operator \* is usually referred to as **Kleene star**. The Kleene star of a set, i.e., an alphabet Σ or a language L, is the set of all words that can be formed by an arbitrary concatenation of the particular symbols plus the **empty word** ε (Ebbinghaus et al., 2013). Suppose we have an alphabet Σ={a} containing the symbol a. The Kleene star of Σ, denoted as Σ\*, contains the empty word ε, the word ε ∘ a = a, a ∘ a = aa, and all other concatenations of the symbol with arbitrary length, which means Σ\*={ε,a,aa,aaa,…}. For the alphabet Σ={a,b}, the Kleene star is Σ\*={ε,a,b,aa,ab,ba,bb,aaa,aab,…}. Besides the Kleene star \*, there is also the Kleene plus +, which is defined as the Kleene star without the empty word: This means that for our alphabet Σ={a,b}, we obtain Σ+={a,b,aa,ab,ba,bb,aaa,aab,…}. The same can be applied to a language L: If the language L={aa, bb}, then L\*={ε, aa, bb, aaaa, aabb, bbaa, bbbb, aaaabb, …} and L+={aa, bb, aaaa, aabb, bbaa, bbbb, aaaabb, …}.

**Empty word**

This is a string of zero length, also known as empty string.

Formal languages are distinguished based on the particular type of grammar that is used to generate them. Thus, a grammar can be thought of as a “language generator.” A central part of the theory of formal languages is the investigation of decision problems associated with formal languages and grammars. Generally, four main problems can be distinguished, which are briefly explained below (Vossen & Witt, 2000).

1. Problem of word: Given a language L and a word w, does the word w belong to the language L, and, thus, is w ∈ L?
2. Problem of equivalence: Given two descriptions of formal languages L1 and L2, do L1 and L2 describe the same language, and, thus, does L1= L2?
3. Problem of emptiness: Given a description of a formal language L, does L contain at least one word or is it equal to the empty set ∅?
4. Problem of infinity: Given a description of a formal language L, does L contain a finite set of words or is the set infinite?

As mentioned before, the concept of (formal) grammars is an important part of the theory of formal languages. A closer look at the definition of a formal grammar can be found below (Révész, 1991).

A formal grammar is defined as the tuple (N, Σ, P, S) and consists of …

… a finite set N of nonterminal symbols.

… a finite set Σ of terminal symbols disjoint from N.

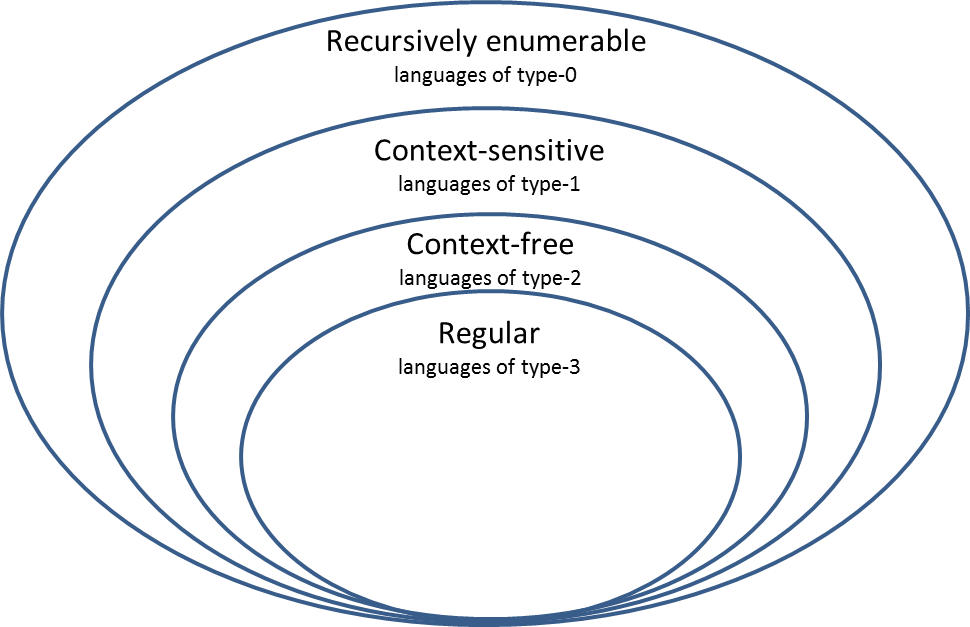
… a finite set of production rules P of the form l→r, where

… a start symbol (also known as sentence symbol) S∈N*.*

Thus, to describe a formal grammar we only need two disjoint sets of symbols N and Σ, a start symbol, and a set of production rules that may seem confusing at first. Simply put, each production rule maps from one string of an arbitrary number of symbols (the head) to another string of an arbitrary number of symbols (the body). The head has to contain at least one nonterminal symbol to be valid.

There are generally four types of formal grammars that play a role in the theory of formal languages. They are defined in the Chomsky hierarchy, which is a hierarchy of classes of formal grammars describing the generation of formal languages (Hunter, 2020). The hierarchy levels are distinguished based on the degree of restrictions for the form of valid production rules P and range from type-0 to type-3 grammars—from completely unrestricted to the highest degree of restrictions. A language L that is generated by a grammar of type-k∈{0,…,3} is called language of type-k, i.e., L∈Lk.

Chomsky Hierarchy



The Chomsky hierarchy, illustrated in the figure above, also represents the set inclusions of the classes of formal languages generated by the different types of grammars. This means that every context-sensitive language is also recursively enumerable, but not vice versa. Formally, the relation L3⊂L2⊂L1⊂L0 holds for the classes of languages generated by the particular formal grammars. In the following, we will present a brief overview of the different types of formal grammars and their restrictions for the form of production rules.

**Turing machines**

This is a model of the operation of a computer and consists of an infinitely long tape.

* Phrase structure grammars (type-0): This class of grammars includes all types of formal grammars without any restriction for the production rules P, thus, a grammar G of type-0 is described by G=(N, Σ, P, S). Formal languages generated by phrase structure grammars are called recursively enumerable languages and they can all be recognized by a **Turing machine** (Copeland, 2004).
* Context-sensitive grammars (type-1): The production rules for context-sensitive grammars are of the form P=uAw→uvw, where A∈N and u,v,w∈(Σ∪𝑁)\*. Furthermore, 𝑆∉𝑢, and v≠ε. All formal languages described by type-1 grammars can be recognized by **linear bounded automata** (Kuroda, 1964).

**Linear bounded automata**

This is a Turing machine which, during calculation, does not leave the range on the tape where the input is located.

* Context-free grammars (type-2): The production rules for context-free grammars are of the form 𝑃=𝑙→𝑟, where l∈N and 𝑟∈(Σ∪𝑁)\*. All formal languages described by type-2 grammars can be recognized by nondeterministic **pushdown automata** (Autebert et al., 1997).

**Pushdown automata**

This is an extension of finite automata by a stack that can be manipulated.

* Regular grammars (type-3): Regular grammars can be divided into left and right regular grammars. The general production rules are restricted to the form 𝑃=𝑙→𝑟, where 𝑙∈𝑁 for both left and right regular grammars, while 𝑟∈Σ∪𝑁Σ and 𝑟∈Σ∪Σ𝑁 for left and right regular grammars, respectively. All formal languages described by type-3 grammars can be recognized by finite automata.

As you have seen above, to each class of formal language corresponds a type of automaton that **recognizes** that very language. An automaton is a model of a behavior consisting of states, state transitions, and actions. While a grammar acts as a generator of a language, an automaton acts as an acceptor (Salomaa, 2014).

**Recognized** **language**

Thisconsists of all the words whose final state is an accepting state.

The theory of automata deals with the classification of different automata on the basis of the available resources. If it is always possible to determine the successor state from the current state—as well as the input character—an automaton is called deterministic. However, if there is a certain degree of freedom in the choice of state transitions and an automaton is allowed to perform a transition from the same pair of current state and input to multiple possible successor states arbitrarily, it is called nondeterministic. The following table shows the relationships between formal languages, their grammars, and the corresponding automata in deterministic and nondeterministic form.

Relationships between Class of Formal Language, Grammar, and Automaton

|  |  |  |  |
| --- | --- | --- | --- |
| **Class of formal language** | **Grammar according to Chomsky hierarchy** | **Deterministic automaton** | **Nondeterministic automaton** |
| Recursively enumerable | Phrase structure grammars (type-0) | Deterministic Turing machine | Nondeterministic Turing machine |
| Context-sensitive | Context-sensitive grammars (type-1) | Deterministic linear bounded automaton | Linear bounded automaton |
| Context free | Context-free grammars (type-2) | Deterministic pushdown automaton | Pushdown automaton |
| Regular | Regular grammars (type-3) | Deterministic finite automaton | Nondeterministic finite automaton |

Finite automata play an important role in practice. For example, in the field of software production they are mostly used to process strings. One of their main purposes is the application of parsers in compilers in order to read and structure input data or to translate between different types of syntax. However, the field of application is not restricted to theoretical and practical computer science: Finite automata are also used within production facilities, for example, for the control of flexible manufacturing systems (Alyoukhin & Silaev, 2019; Darabi et al., 2003; Kobetski & Fabian, 2009).

A finite automaton is characterized by the fact that the set of possible states is finite, which is why it is also known as a finite state machine. Formally, one can define a finite automaton acting as an acceptor as shown below (Salomaa, 2014).

A finite automaton is described by a tuple A={K,Σ,δ,s0,F}, where

K is a finite set of states,

Σ is an alphabet,

δ is a total transition function K×Σ→K, thus defined for each possible input,

s0∈K is the initial state, and

F⊆K is a set of final states.

Since the relationship between finite automata and formal languages was only explored superficially until now, we need to specify that regular languages accepted by a finite automaton are those described by regular expressions. The definition of a regular expression is shown below (Chakraborty, 2003).

Given an alphabet Σ…

… the empty set ∅ is a regular expression.

… each symbol a∈Σ is a regular expression.

Given the two regular expressions a,b∈Σ,

(a|b) (alternative),

ab (*c*oncatenation), and

a\*

are regular expressions.

An easy example for a regular expression is R=a(b|c)\*a. If the expression starts with the symbol a followed by an arbitrary number of concatenations of the symbols b and c and ends with the symbol a, it means that a word w belongs to a regular language L described by R. Thus, the words w1=abca, w2=abba,and w3=accba are elements of L described by R, while w4=abc, w5=bba, and w6=aaa are not. If they accept the same language, two automata are called equivalent.

Given that only regular languages can be recognized by finite automata, it is often necessary to prove or disprove the regularity of a formal language. A useful approach for this particular task is the pumping lemma (more on this in Lawson, 2003).

## 2.2 Deterministic Finite Automata

As mentioned previously, a deterministic finite automaton is characterized by the fact that it is clearly determined in every situation. Let us have a look at an example to see how this property works. Suppose that we want to create an acceptor for the regular expression a(b|c)\*a. The automaton can be fully described by the tuple A={K,Σ,δ,s0,F}, where

K={s0,s1,s2}

Σ={a,b,c}

s0= s0

F={s2}

δ: δ(s0,a)= s1

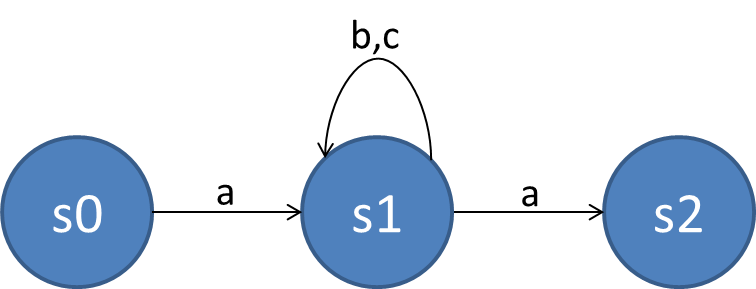
δ(s1,b)= s1

δ(s1,c)= s1

δ(s1,a)= s2.

The following figure shows the corresponding automaton.

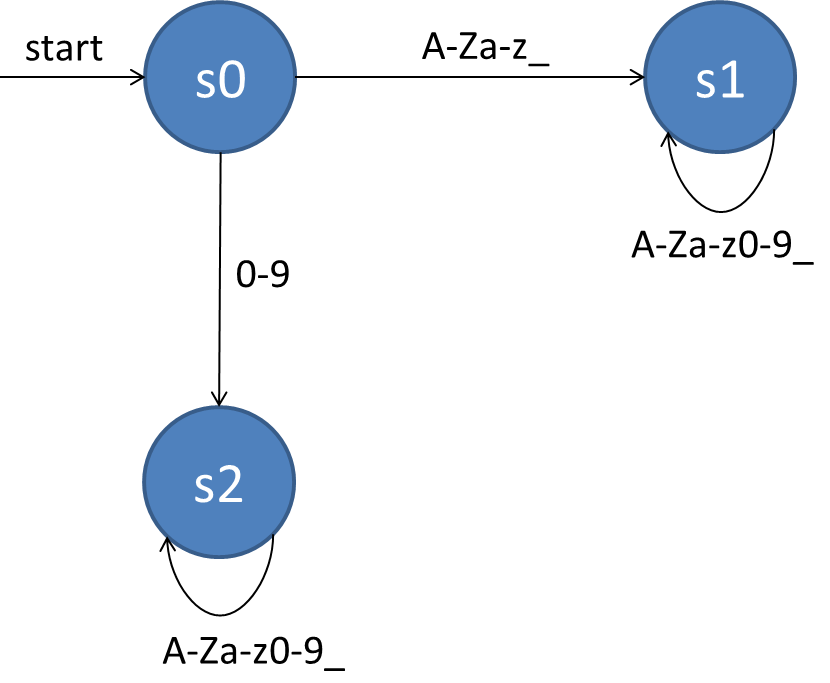
Deterministic Finite Automaton for a(b|c)\*a



The defined automaton contains four transition functions between states, which means that the number of arrows equals the number of transition functions of δ (the arrow for a,b can be separated into two single ones). In order to see the functionality of the transition of states, let us see what happens when we enter the word abcba. The initial state s0 is the only one left when the character a is entered. Since our test word starts with an a, the automaton transits to state s1. Now, to change the state, there are three possibilities. The regular expression expects an arbitrary number of combinations of b and c following an a. Thus, the automaton stays in state s1 as long as either b or c is entered. The Kleene star of the alternative also allows a direct transition from s1 to the final state s2 without any b or c. In our case, we stay in state s1 for three characters after the initial a and finish the word by an a leading to the only possible final state s2. As you can see, each possible input leads to a clearly defined successive state: This is the determinism of the automaton. There is no situation in which we do not know what happens after entering a certain character.

It is important to remember that a word is accepted by an acceptor if and only if one of its final states is reached. If the input stops before that or it reaches a state that cannot be left anymore, the input is considered rejected. This is why it is especially important for parsers in compilers to check the names of variables. For example, in Python, a variable has to start with a letter or an underscore character and can only contain alphanumeric characters and underscores. Thus, the underlying alphabet is Σ={A,B,…,Z,a,b,…,z,\_,0,1,…9} and the corresponding deterministic finite automaton can be represented as shown below.

Acceptor for Variable Names in Python



As you can see, the initial state is denoted by s0. There are two possible successive states, but only s1 is final. Entering a letter or an underscore as the first character leads to the final state, which allows you to insert an arbitrary number of characters chosen from the whole alphabet. However, if a number is entered as the first character, the automaton transits to state s2, which can never be left regardless of what the following character is. In this case, any word entered is rejected and therefore does not constitute a valid Python variable.

For every deterministic finite automaton there is a unique minimal automaton that accepts the same formal language (Hopcroft et al., 2001). Generally, two types of states can be removed from an automaton: unreachable and indistinguishable states. Different approaches exist to perform a minimization, such as Hopcroft’s algorithm (Hopcroft, 1971), Moore’s algorithm (Moore, 1956), and Brzozowski’s algorithm (Brzozowski, 1962). For a more detailed explanation of these algorithms, the interested reader is referred to the literature.

Besides acceptors there is another type of finite state machines that plays an important role in various applications—the transducer. As the name suggests, the transducer “transforms” an input into a certain output and is defined as shown below (Srimani & Nasir, 2007).

A deterministic finite state transducer is described by a tuple T={K,Σ, Γ,δ,s0,F}, where

K is a finite set of states,

Σ is an input alphabet,

Γ is an output alphabet,

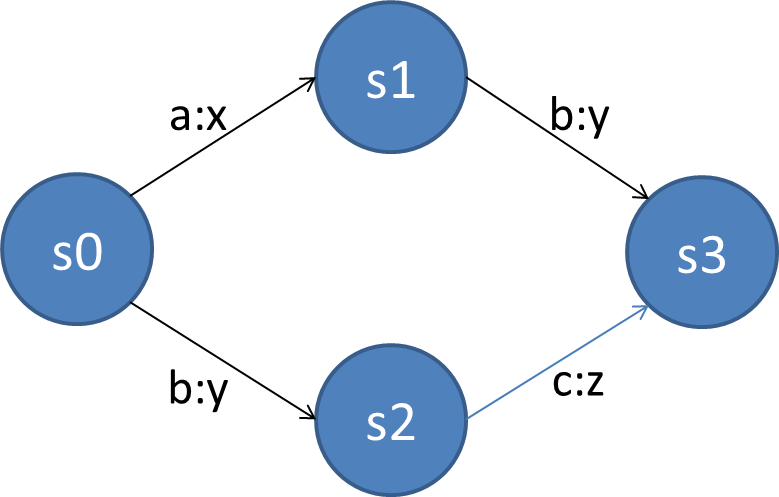
δ is a total transition function K×Σ→K, thus defined for each possible input,

S⊆K is the set of initial states, and

F⊆K is a set of final states.

While an acceptor accepts or recognizes strings and maps them to the set {0,1} for rejecting and accepting, respectively, a transducer generates strings and thereby a formal language, i.e., a set of words used for mathematical purposes. This means that an input string is either accepted and translated to a new one or it is rejected and no output is produced. The following figure shows a very simple example of a transducer in order to better represent its functionality.

Example of Deterministic Finite State Transducer



In this example, the set of states K={s0, s1, s2, s3}, the input alphabet Σ={a,b,c}, the output alphabet Γ={x,y,z}, the set of initial states S={s0}, and the set of final states F={s3}. The transition function δ describes the following transitions:

δ(s0,a)= s1,x

δ (s0,b)= s2,y

δ(s1,b)= s3,y

δ(s2,c)= s3,z.

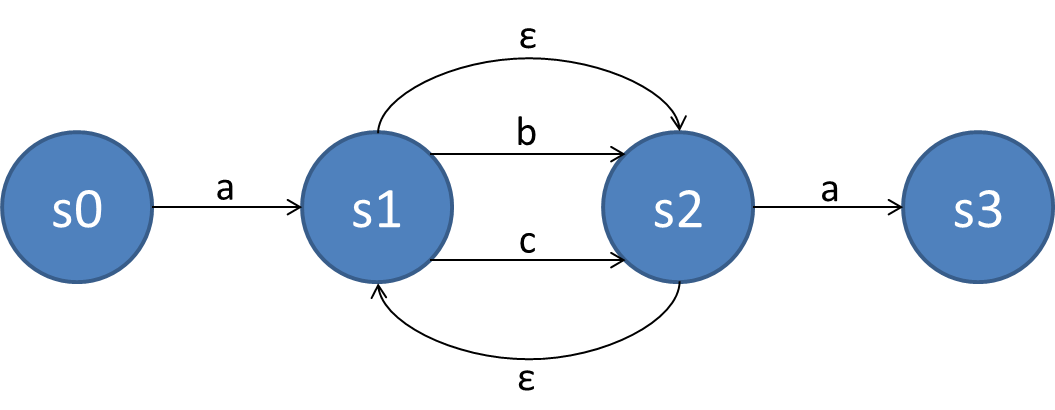
We see that the transition function defines both a successive state and an output that results from a particular pair of input and current state. Our exemplary finite state transducer translates the strings ab to xy and bc to yz. This is, of course, only a very simple demonstration but it can easily be extended to transducers that perform tasks such as data encoding or data compression.

## 2.3 Nondeterministic Finite Automata

Just as the deterministic finite automaton is characterized by its complete determinism, the nondeterministic variant is characterized by the fact that it does not fulfill that criterion. There is no unique successive state to transition to from a previous one at a certain input. In other words, there are multiple equivalent possibilities for the transition and it is impossible to determine which one is chosen by the automaton at any given time (Salomaa, 2014). Furthermore, the empty word ε is a legal input to change between states and therefore originates a transition called ε-transition.

The difference between the formal definition of a deterministic and a nondeterministic finite automaton lies in the description of the transition. While the former defines δ as a transition function, the latter defines it as a transition relation. This difference has a strong impact on the behavior of the particular automaton. Let us have a look at the regular expression a(b|c)\*a and see how we can create a nondeterministic version of this particular acceptor. The following figure shows the corresponding automaton.

Nondeterministic Finite Automaton for a(b|c)\*a

**

The automaton can be fully described by the tuple A={K,Σ,δ,s0,F}, where

K={s0,s1,s2,s3}

Σ={a,b,c}

s0= s0

F={s3}

δ: δ(s0,a)= s1

δ(s1,b)= s2

δ(s1,c)= s2

δ(s2,a)= s3

δ(s1,ε)= s2

δ(s2,ε)= s1.

Compared to the deterministic variant, we have now added two ε-transitions. When we now enter the word abcba, the initial state s0 is, again, the only one left when the character a is entered. Since our test word starts with an a, the automaton transits to state s1. Now there are three possibilities as to how the state can be changed. The regular expression expects an arbitrary number of combinations of b and c following an a. Thus, the transition from s1 to s2 happens by entering either b or c. The Kleene star of the alternative adds a third possibility—the empty word ε. In our case, the next character is a b, which changes the state to s2. Now, a state transition can happen either if an a is entered, triggering a transition to s3, or if b or c is entered, changing the state back to s2 by means of an ε-transition. The transitions between s1 and s2 show the recursive structure of our regular expression. In our case, we transit three times from s1 to s2 and finish the word by an a leading to the only possible final state s3.

Every nondeterministic finite automaton can be transformed into a deterministic one by means of a **powerset construction** (Rabin & Scott, 1959). The basis for that statement is the fact that every formal language that is accepted by a nondeterministic acceptor is also accepted by a deterministic one. In the following, we want to present how this construction works in principle.

**Powerset construction**

This is important to prove that a nondeterministic finite automaton is not capable of recognizing different languages than deterministic finite automata.

The aim is to construct an equivalent deterministic automaton defined by the tuple A={K’,Σ’,δ’,s0’,F’} for a nondeterministic automaton defined by NA={K,Σ,δ,s0,F}.

1. We start with empty sets of states K’ and F’.
2. The initial state of A is chosen to be a singleton s0’={s0} and s0’ is added to the set of states K’.
3. The same must be done for all states k’ that are already elements of K’.

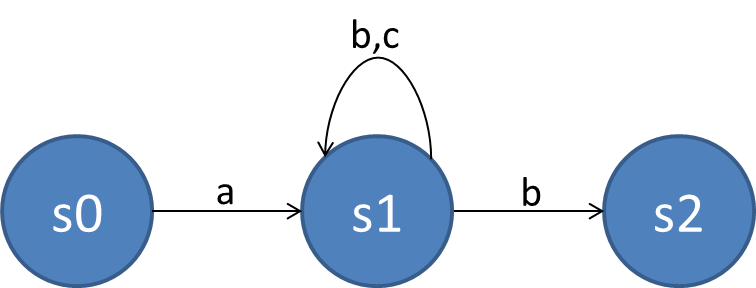
For all input characters σ∊Σ:

* + 1. Construct a successive state k’’ as the set of all states that NA can transit to by entering σ. Formally, k’’≔⋃{δ(r,σ)│r∈k’}.
    2. Add k’’ to K’ if k’’∉ K’.
    3. Add δ’(k’, σ)=k’’ to the transitions of A.

1. Repeat step 3 until K’ and δ’ do not change anymore.
2. Choose the set of final states F’ of A to be a subset of K’ that contains a final state of F.

According to the Myhill–Nerode theorem, a constructed deterministic finite automaton A can consist of 2|K| states, where K is the set of states of the nondeterministic automaton (Nerode, 1958). Thus, regular expressions that are captured by a nondeterministic automaton can produce complex deterministic equivalents in a relatively compact way, e.g., an NA with 10 states can result in 210=1024 states in the worst case. Let us have a look at an example for the presented conversion considering the regular expression a(b|c)\*b. The following figure shows the nondeterministic automaton NA.

Nondeterministic Finite Automaton for a(b|c)\*b

**

The nondeterminism of the automaton can be seen clearly. The successive state of s1 after entering the character b is not clearly defined and can either be s1 or s2. The transition δ can be represented by

δ(s0,a)=s1,

δ(s0,b)= ∅,

δ(s1,b)={s1,s2},

δ(s1,c)=s1.

We start by choosing the singleton s0’={s0} and adding s0’ to K’ of A. Then, we add all the successive states of NA from the state s0 to K’ and supplement the transition function δ’ with the particular state transitions. Thus, δ’ consists of

δ’(s’0={s0},a)=s’1={s1},

δ’(s’0={s0},b)=0=∅

and K’={s’0={s0}, s’1={s1},0} after the first iteration. Finally, we construct the successive state s’2={s1,s2} and add it to K’. The corresponding transitions are given by

δ’(s’1={s1},a)=s’1={s1},

δ’(s’1={s1},b)=s’2={s1,s2},

δ’(s’2={s1,s2},b)=s’2={s1,s2},

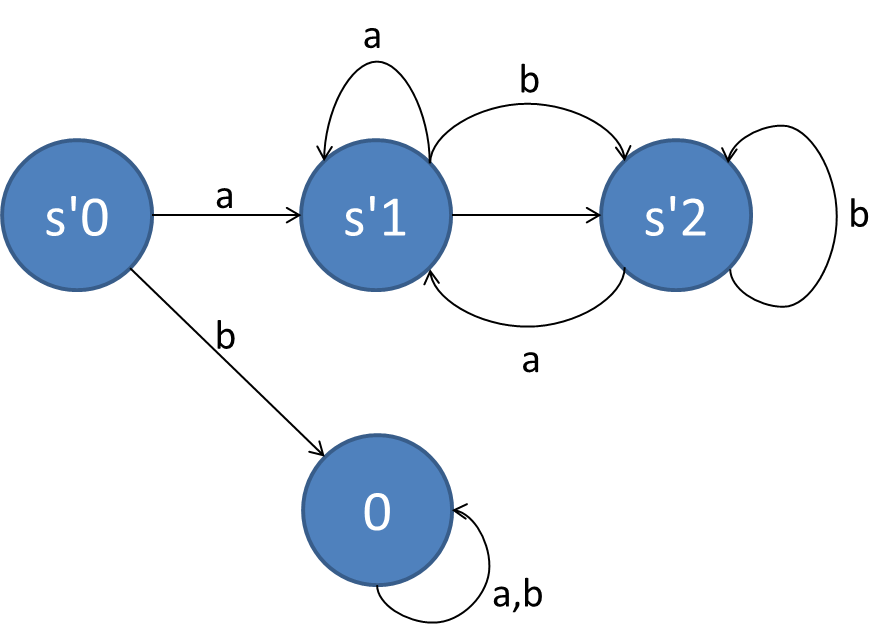
δ’(s’2={s1,s2},a)=s’1={s1},

δ’(0=∅,a)= 0=∅,

δ’(0=∅,b)= 0=∅.

In doing that, the algorithm ends and we can draw the constructed deterministic finite automaton as shown in the following figure.

Deterministic Finite Automaton for a(b|c)\*b



The construction of an equivalent deterministic automaton is relatively easy for simple, nondeterministic automata, but it can become significantly more complex for larger ones (for more on this topic, see Hopcroft et al., 2001; Rabin & Scott, 1959).

## 2.4 Properties

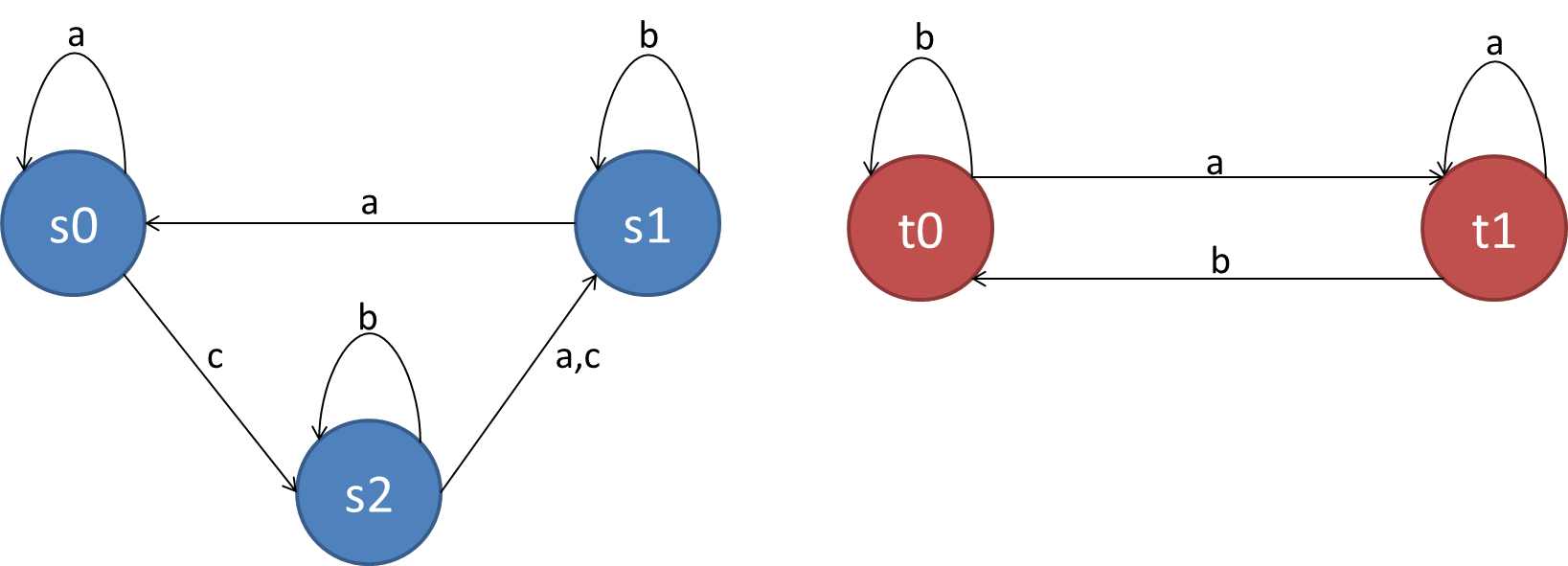
So far, we introduced the general structure of finite automata and presented the deterministic and nondeterministic variants in detail by means of several examples. The general properties of a finite automaton can be summarized as follows: Finite automata and regular expressions define regular languages. More precisely, the regular language that is defined by a regular expression is exactly the language that is also accepted by a finite automaton.

Finite automata can be combined by means of different composition operations, where two different types are generally considered: the parallel composition denoted as || (synchronous composition) and the product denoted as × (completely synchronous composition) (Cassandras & Lafortune, 2009). The two compositions can be better explained by considering the two finite automata A1={K1,Σ1,δ1,s01,F1} and A2={K2,Σ2,δ2,s02,F2}. The product of A1×A2 is defined as

where Ac is the accessible part of the resulting automaton, thus consisting only of the states that can be reached. K1×K2 and F1×F2 stand for the combination of the sets of states, e.g., if K1={a,b} and K2={x,y} then K1×K2={a.x,a.y,b.x,b.y}. The transition function is defined as

In other words, if a transition is realized by entering character σ for state k1 of A1 and k2 of A2, then there will be a new transition for the combined state k1.k2. The alphabet of the resulting automaton consists of the intersection of both alphabets, i.e., of those characters that exist in both alphabets. Let us have a look at an example for that composition technique. Consider the two finite automata shown in the following figure.

Automaton A1 (Left with Blue States) and A2 (Right with Red States)



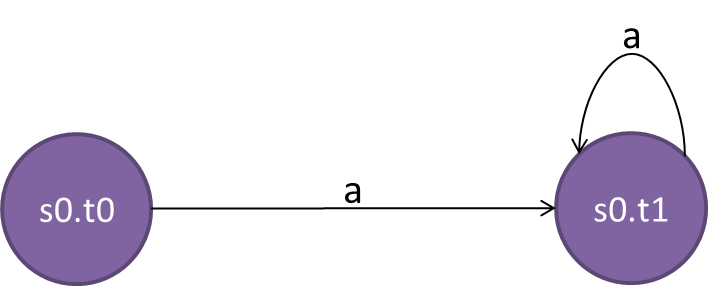
To determine the product of A1 and A2 we first combine the sets of both states K1×K2, which yields {s0.t0,s0.t1,s1.t0,s1.t1,s2.t0,s2.t1}. The set of final states yields F1×F2={s1.t1} and the resulting alphabet consists of the characters {a,b}, because c is not included in A1. The initial state is the combination s0.t0. Finally, we determine the transition function, which yields

δ(s0.t0,a)=s0.t1,

δ(s0.t1,a)=s0.t1.

Thus, the product of A1 and A2 can be represented as shown in the following figure.

Product Composition of A1 and A2



The parallel composition of two finite automata is defined as

In comparison to the product composition, the resulting alphabet consists of the union of the alphabets of A1 and A2. Furthermore, the transition function is defined as

Let us visualize the parallel composition by means of the two automata from the previous example. The set of states resulting from the initial state and the set of final states is equivalent to the product composition. The alphabet is the union of both alphabets, in this case, Σ={a,b,c} and the transition function yields

δ(s0.t0,a)=s0.t1,

δ(s0.t1,a)=s0.t1,

δ(s0.t0,c)=s2.t0,

δ(s0.t1,c)=s2.t1,

δ(s2.t0,b)=s2.t0,

δ(s2.t0,a)=s1.t1,

δ(s2.t1,a)=s1.t1,

δ(s2.t0,c)=s1.t0,

δ(s2.t1,c)=s1.t1,

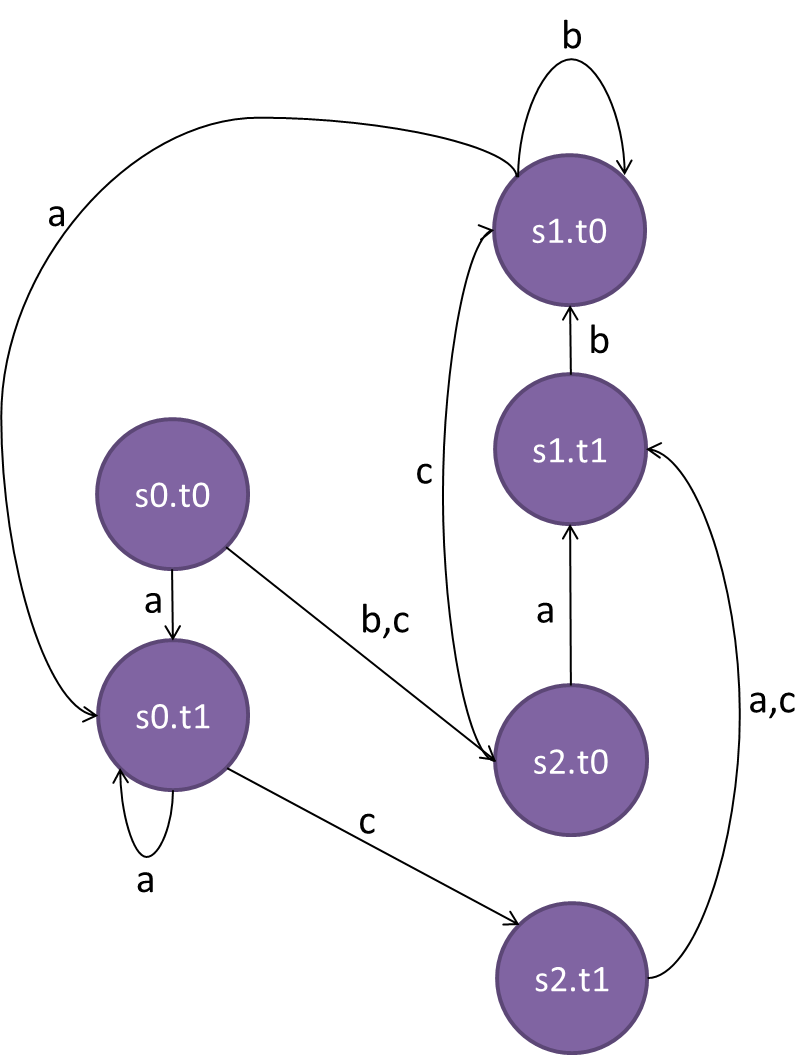
δ(s1.t0,b)=s1.t0,

δ(s1.t0,a)=s0.t1,

δ(s1.t1,b)=s1.t0.

The following figure shows the automaton resulting after parallel composition.

Parallel Composition of A1 and A2



An important remark to make is that parallel composition does not affect the state transitions of A1 triggered by entering a character that is not an element of the alphabet of A2. Furthermore, if Σ1=Σ2, then the parallel equals the product composition. Another thing that must be considered is that the product composition is commutative while the parallel composition is not, i.e., A1×A2= A2×A1 but A1||A2≠ A2||A1.

|  |
| --- |
| Summary |
| In contrast to natural languages, formal languages are abstract models of languages that serve a mathematical purpose. They consist of a set of words over an alphabet, which is a set of possible symbols or characters used to describe the handling of strings.  Formal grammars generate formal languages by applying certain production rules on the basis of an initial symbol and can be classified from type-0 to type-3 by means of the Chomsky hierarchy. Type-0 grammars are called phrase structure grammars, they do not restrict the possible production rules, and they are accepted by Turing machines. Type-1 and type-2 grammars are called context-sensitive and context-free, respectively, and are accepted by linear bounded automata and nondeterministic pushdown automata with an increasing degree of restrictions for the production rules. Finally, type-3 grammars are called regular grammars and are accepted by finite automata.  Finite automata can be deterministic or nondeterministic. Every nondeterministic automaton can be transformed to a deterministic equivalent by means of powerset construction. While the successive states of a deterministic automaton can clearly be determined based on the current state, there are multiple possible successive states for a nondeterministic one. There are different types of composition techniques for finite automata, of which the product and the parallel composition are only two examples. |

# Unit 3 – Petri Nets

**Study Goals**

On completion of this unit, you will have learned …

… how Petri nets are formally defined and what different variants exist.

… how Petri nets can be used to model systems.

… the properties of Petri nets.

… the purpose of reachability graphs.

… what T- and P-invariants are and how they can be used to analyze Petri nets.

# 3. Petri Nets

## Introduction

Petri nets are discrete, distributed models that are often used to describe the dynamic behavior of systems. Based on finite automata, Petri nets were developed in the 1960s to describe concurrency. Nowadays, they are not only used in computer science, but also in theoretical biology (Chaouiya, 2007; Heiner et al., 2008), business processes (van Der Aalst, 2002; Lohmann et al., 2009), logistics (Macías & de la Parte, 2004), and engineering (DiCesare et al., 1993). In this section, we will present the basics and mathematical description of this useful technique and how it can be used for the modeling of systems. Furthermore, we will take a closer look at some of the special properties of Petri nets and how their behavior can be analyzed.

## Preliminaries

The basic version of a Petri net is represented by a graph with two types of vertices called places and transitions (Peterson, 1977).

A graph G is an ordered pair (V,E) of a set of vertices V and a set of edges E⊆V×V.

By looking at the structure of finite automata, it becomes clear that those are graphs and that they consist of states and state transitions (the vertices and edges, respectively). The special thing about the graphs used for Petri nets is that two different types of vertices exist. In graph theory, that variant is called a **bipartite** graph (Bollobás, 2012).

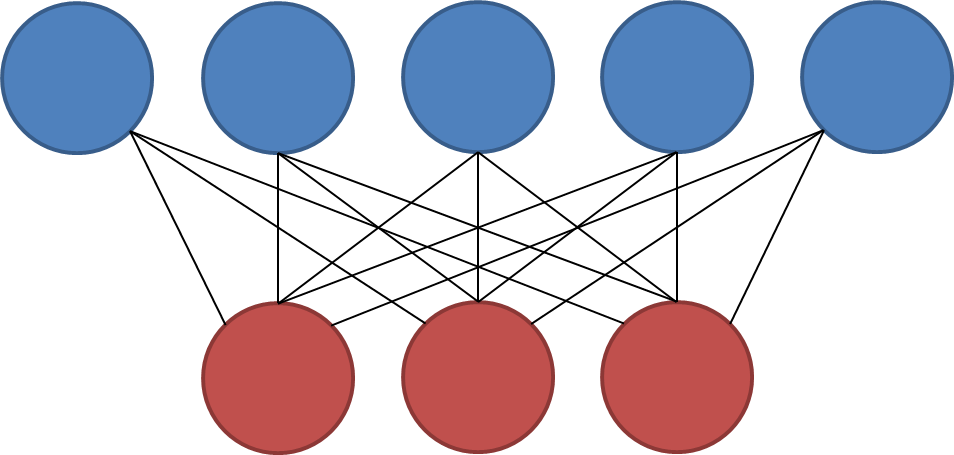
**Bipartite**

This means that something consists of two parts. Another name for a bipartite graph is bigraph.

A bipartite graph is a graph with two disjoint and independent sets of vertices U and V and is represented by the triple G=(U,V,E). An edge always connects a vertex in U with one in V.

To illustrate this formal definition, we explain the components by means of the following figure.

Complete Bipartite Graph



The bipartite graph shown above is complete because the set of edges E (black lines) defines a connection between each ui∈U (blue circles) and each vj∈V (red circles), where i∈[1,m] and j∈[1,n]. Thus, a complete bipartite graph has m∙n edges.

A Petri net is a bipartite graph in which the two different types of vertices—called places and transitions—describe states and state transitions (or activities), respectively. Formally, it can be defined as follows (Peterson, 1977).

A Petri net is a bipartite graph described by the triple (P,T,F), where

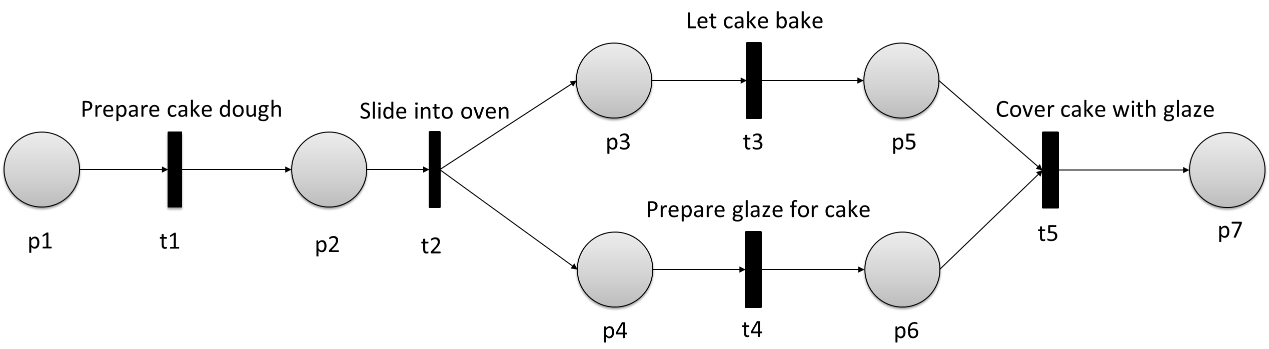
P is a finite set of places represented by circles,

T is a finite set of transitions, where T∩P=∅, represented as long narrow rectangles, and

F is a set of flow relations F⊆(P×T)∪(T×P), also called arcs, represented as arrows.

To provide an example, we can model the process of baking a cake through the following Petri net.

Petri Net for Cake Baking Process

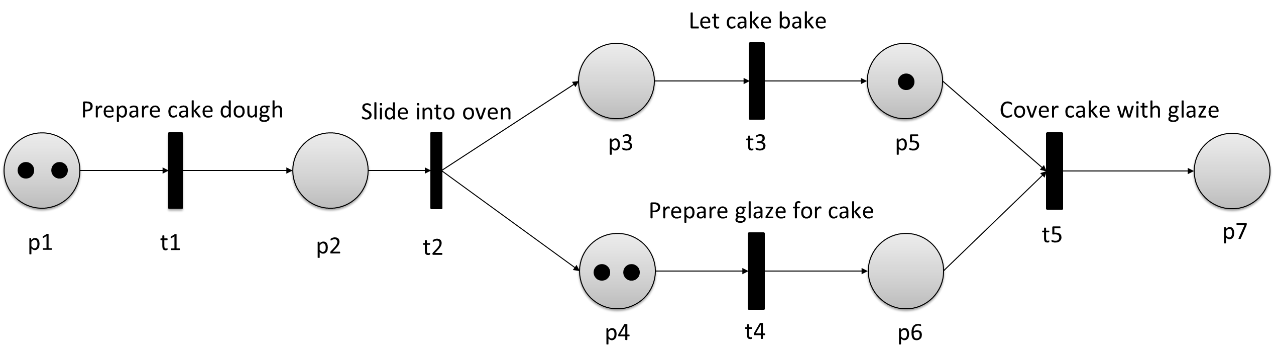


This looks like a finite automaton, with the difference that this has two different types of vertices. Here, the net has to be brought to life, i.e., we have to model its dynamic behavior. This is done by assigning a certain number of tokens to each of the places, which represents the particular amount of available resources required for a transition. The mapping relating the places to the number of tokens is called the marking of the Petri net.

A marking of a Petri net is defined by the mapping M: P→ℕ0.

A marking of the previously shown Petri net could look like the one in the figure below.

Marking of a Petri Net



The dynamic behavior is the result of a switching of the transitions and the exchange of tokens, commonly known as a token game. To explain this process, we introduce the following definitions of preset and postset (Gold, 2004).

The preset ●x of a vertex x∈P∪T is the set of vertices that are the origin of an arc to x, thus, ●x={y|(y,x)∈F}. The postset x● of a vertex x∈P∪T is the set of vertices that are the destination of an arc from x, thus, x●={y|(x,y)∈F}.

The preset and postset of a transition consist of places and are called input and output places, respectively. The same relation is also valid for places, thus, their preset and postset consist of transitions, called input and output transitions.

The initial marking of a Petri net defines the number of tokens for each particular place. A transition is defined as enabled if and only if each of the input places contains at least one token. In this case, the transition is able to fire, i.e., it consumes one token from each input place and produces one token at each output place. By firing, a transition changes the marking of a Petri net. If a certain transition t∈T is able to change the marking m∈M into the marking m’∈ M, we can write

.

A transition can also fire if ●x=∅, (meaning it can fire at any time), which means it can consume tokens without producing new ones if x●=∅. The whole process can be described as a relocating of resources. Having a look at our previous example with marking m we can identify that

and .

With the subsequent marking, it holds that

, , and .

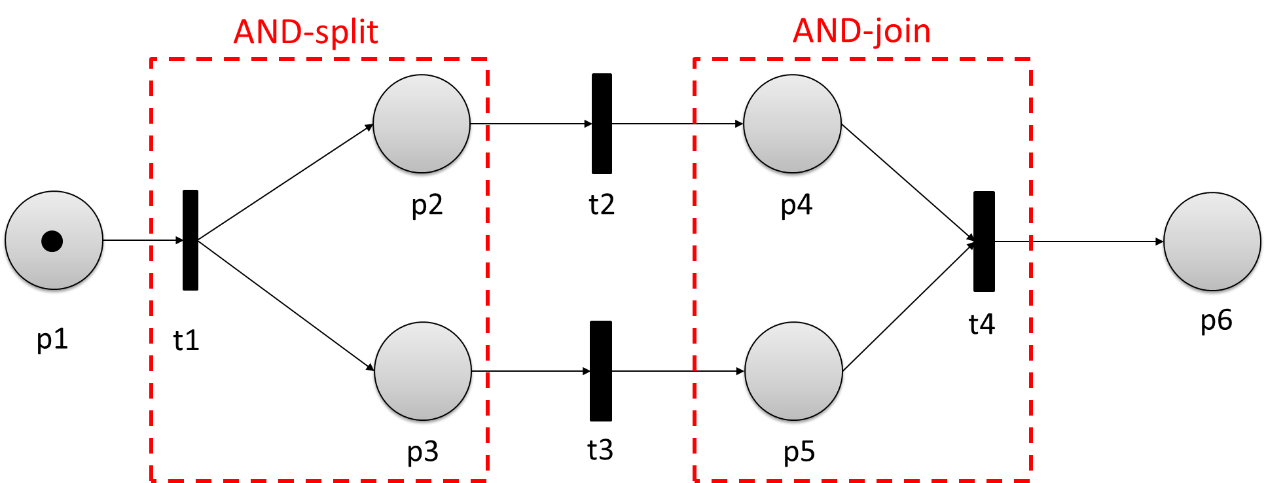
There are generally two types of constructions that define a split of a Petri net: the AND-split and the **XOR**-split. While the former defines a parallel execution, the latter creates race conditions.

**XOR**

The XOR-operator generally means that the values of its inputs have to be different to output the value true.

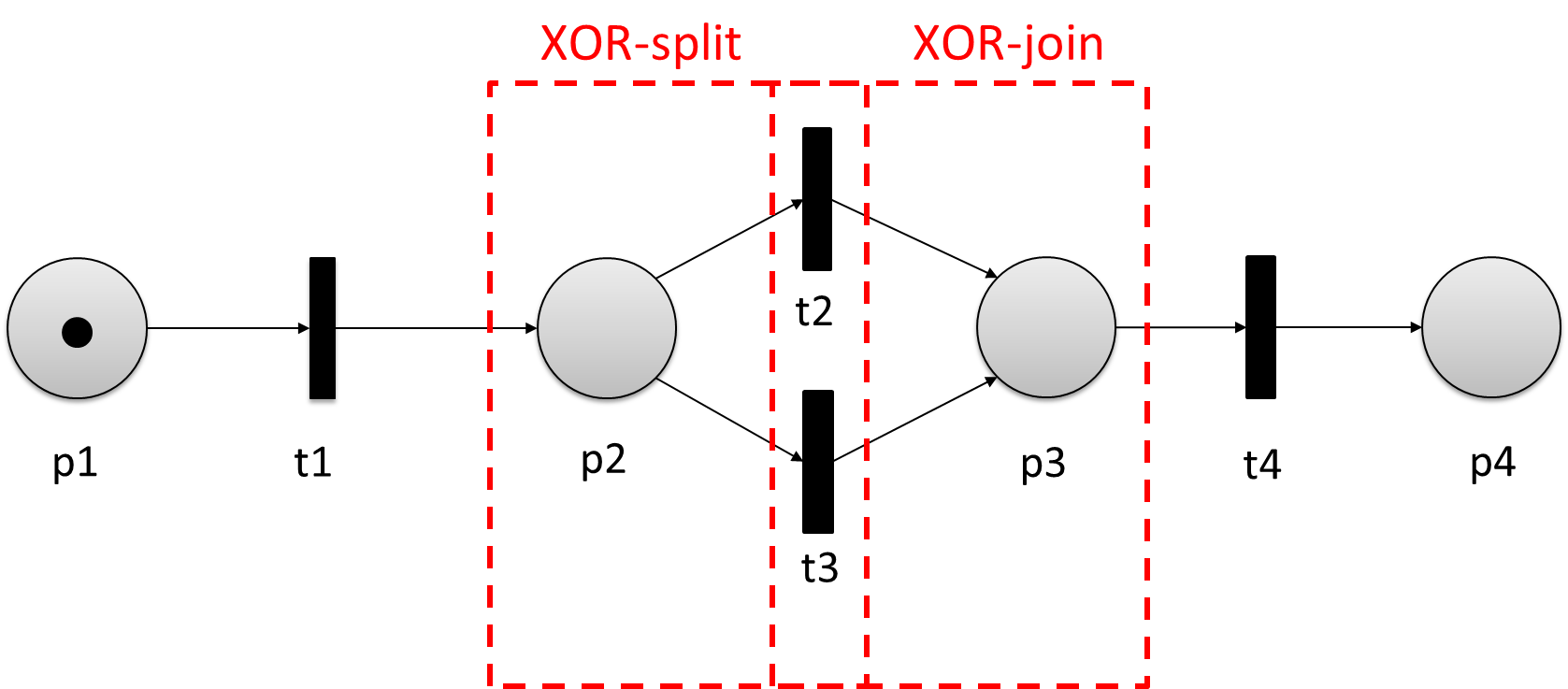
Following a transition, the AND-split is defined by two and more places and, thus, each firing produces new tokens in each place. Conversely, an AND-join is defined by one transition following two or more places. In that case, the transition is enabled if and only if each place is marked with at least one token (Lohmann et al., 2009). The following figure shows an exemplary construction of an AND-split combined with an AND-join.

AND-Split and AND-Join



The XOR-split, instead, is defined by two or more transitions following a place. In that case, only one of the transitions can fire when the place is marked. Conversely, a XOR-join is defined by one place following two or more transitions, where only one of the transitions is allowed to fire (Lohmann et al., 2009). The following figure shows an exemplary construction of a XOR-split combined with a XOR-join.

XOR-Split and XOR-Join



After talking about all the components and the mathematical definitions of general Petri nets, we can now examine their different variants (Reisig, 2012; van der Aalst, 1998; Jensen, 1987).

* Condition-event nets: A condition-event net is defined by the fact that each place can only have a maximum of one token. If a particular place is set with a token, the condition is considered met, which, in turn, leads to an event (the firing of a transition).
* Workflow nets: A workflow net is defined by the fact that it has exactly one initial and one final place xi and xo, respectively, where ●xi =∅ and xo●=∅. Furthermore, each place p∈P and transition t∈T is part of the path from xi to xo.
* Place-transition nets: A place-transition net is a Petri net extended by a weight function W:F→ ℕ, defining the weight of each arc f∈F. A transition becomes enabled and will be able to fire if and only if the numbers of tokens are set at the input places, while it consumes that very amount of tokens and produces as many new ones in the output place as defined by the particular weight.
* Colored Petri nets: A colored Petri net differentiates multiple types of markers, allowing for more complex conditions when transitions can fire.

## 3.2 Modeling Systems

To model a particular discrete system by means of a Petri net, one has to first identify a set of different components that need to be incorporated. The typical components of this set are defined below.

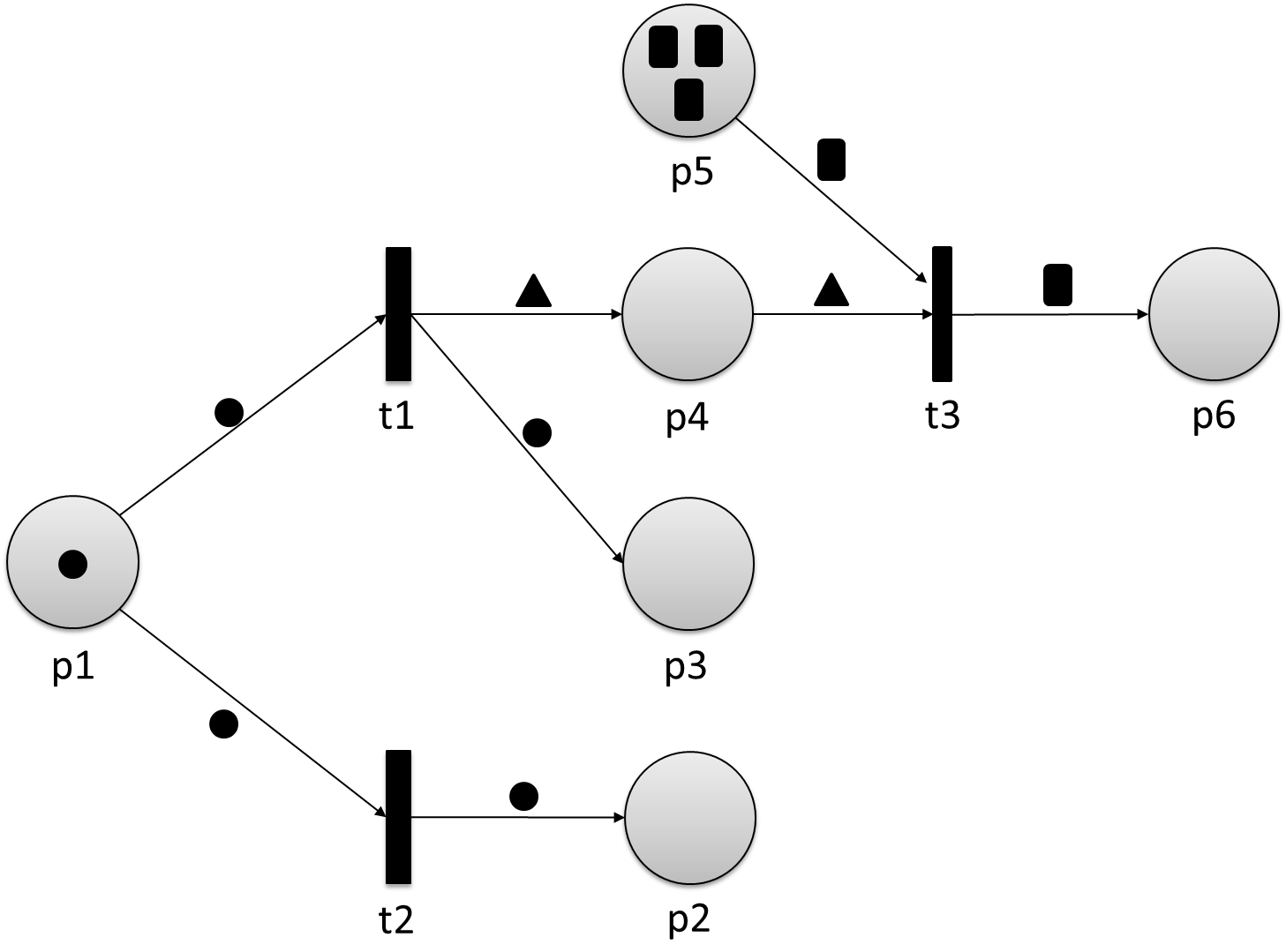
* Storage components: These are components that can contain certain types of resources and that can be equipped with a certain condition. In the Petri net, components are represented by the places.
* Resources: The resources are the elements that are produced, transformed, and consumed in the systems. They are the tokens of the Petri net.
* Activities: Activities are components that can change the contents of the storages. They are represented by transitions and must follow a place.
* States: A state is a “snapshot” of the system with a certain distribution of resources over the storage components. It is represented as a marking of the Petri net, although what is mostly shown is the initial marking, i.e., the initial resource distribution.
* State transitions: State transitions represent the flow between different states of the systems, i.e., the flow between different markings of the Petri net.

Let us have a look at an example of a discrete system that can be modeled by a Petri net. Consider a drinks machine that gives out a can when it is fed with coins. What are the possible storage components for this system? The first one is the slot for the coins p1, followed by the coin return tray p2, the cash register p3, a signal component containing the signal that was given to the system p4, the storage for the cans p5, and the output tray p6. It is obvious that these components can all contain a certain type of resource, i.e.,

* the coin slot, cash register, and coin return tray can contain coins,
* the storage for cans and the output tray can contain cans, and
* the signal storage can contain signals.

Now, after defining the places and types of tokens for the model of the drinks machine system, we have to identify the activities—or transitions—of the model. One activity can be pressing a button to produce a signal representing a can request t1; another example of activity is a return button that returns the inserted coin t2 if the customer decides not to take the can; a third activity could be the output of a certain can into the output tray given a signal t3. How can the flow of the systems be described? Transition t1 will be enabled if a coin is inserted and fires to produce a signal in the signal component and put a coin in the cash register. The return button t2 is also enabled after inserting a coin and fires to move a coin to the coin return tray. For transition t3 we can identify two input places, i.e., the can storage and the signal component. If both are marked with at least one particular token (a signal and a can), t3 is enabled and can fire to move a can to the output tray. The resulting Petri net is illustrated in the following figure, where the tokens for the coins, signals, and cans are represented by circles, triangles, and rectangles, respectively.

Petri Net for Drinks Machine with Initial Marking



This example shows the general modeling process of discrete systems to create a Petri net. Of course, this is a very naïve consideration of the behavior of a drinks machine and multiple aspects were neglected in our example, such as the differentiation between different coins and the selection of different drinks, among others.

## 3.3 Properties

Petri nets are a convenient technique to model certain properties of concurrent systems, such as liveness, deadlock freedom, termination, and boundedness. The formal definitions of these terms are illustrated in the box below (Murata, 1989).

* Liveness of a Petri net: A transition t∈T is called live if there is a sequence Tn of transitions for each reachable marking m, so that

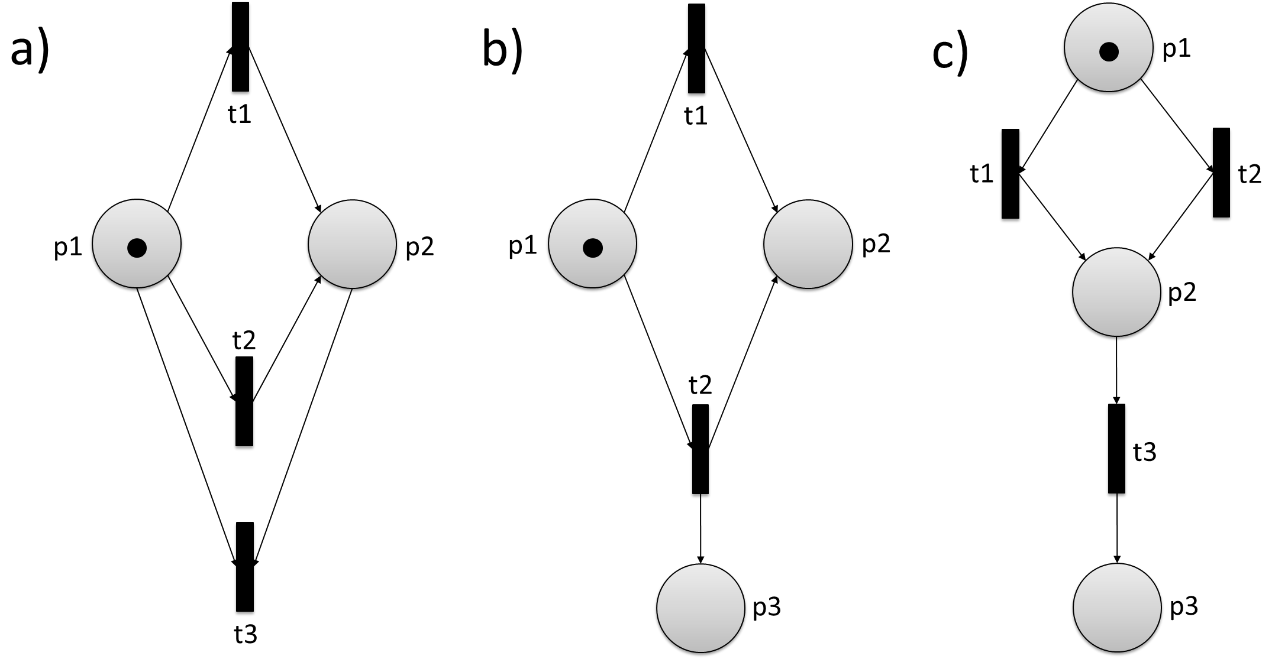
,

where t is enabled in *m’*. The whole Petri net is called live if all of its transitions are live.

* Deadlock freedom of a Petri net: A Petri net is called deadlock free if there is at least one enabled transition for each reachable marking.
* Termination of a Petri net: A Petri net terminates if each sequence of transitions starting in marking m ends in a deadlock at some point.
* A place p∈P is called k-bounded if it is not marked with more than k tokens at any reachable marking (including the initial marking) and generally bounded if it is k-bounded for any k∈ ℕ. If it is 1-bounded, it is called safe. A Petri net is k-bounded, bounded, or safe if each of its places satisfies that property.

We can visualize the different properties by means of the following exemplary Petri nets.

Properties of Concurrent Systems



The Petri net shown in a) is not live because transition t3 will never be enabled and therefore can never fire. All of the places are 1-bounded because at any marking they are marked with a maximum of 1 token. Hence, the Petri net is 1-bounded and consequently safe. It is not deadlock free because there is no enabled transition after the firing of t1 or t2. Since the sequences of transitions over t1 and t2 are the only possible ones and both end in a deadlock, the Petri net terminates.

All of the transitions of the Petri net shown in b) will be enabled and can fire, which means that the Petri net is live. The net is not bounded because the number of tokens in p3 can grow to an unlimited number and it does not terminate because it is deadlock free.

The Petri net in c) does not satisfy the property of liveness. It contains a deadlock after the firing of t2 and t3 but it does not terminate because not every sequence of transitions ends in a deadlock (if t1 and t2 fire alternately). It is bounded and safe because there is no possibility for producing additional tokens.

## 3.4 Analysis Methods

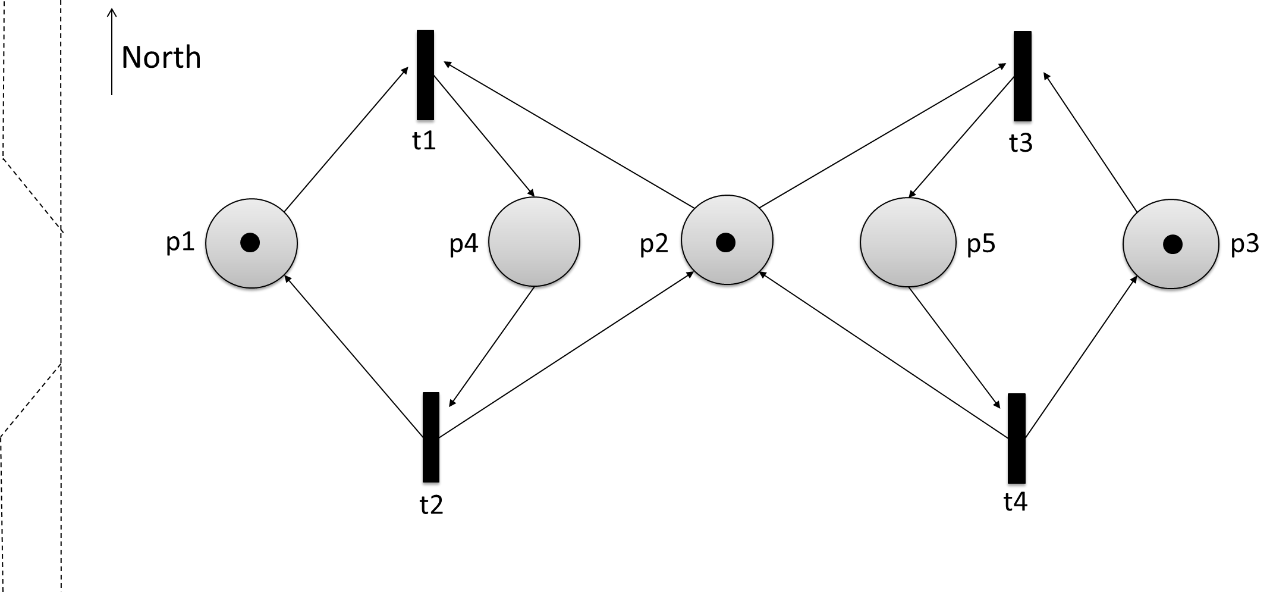
A widely used method to analyze the behavior of Petri nets is the reachability graph. It represents all possible markings of the net and their corresponding transitions and it is a convenient technique if one wants to determine whether certain states are reachable or not. An important basis for that is the reachability set, which is defined below (Popova-Zeugmann, 2013).

reachability set R(m) of a marking m in a Petri net N is the set of all markings m’ that are reachable from m.

The reachability set R(N) of a Petri net N is defined by the set of all markings m’ that are reachable from the initial marking m0.

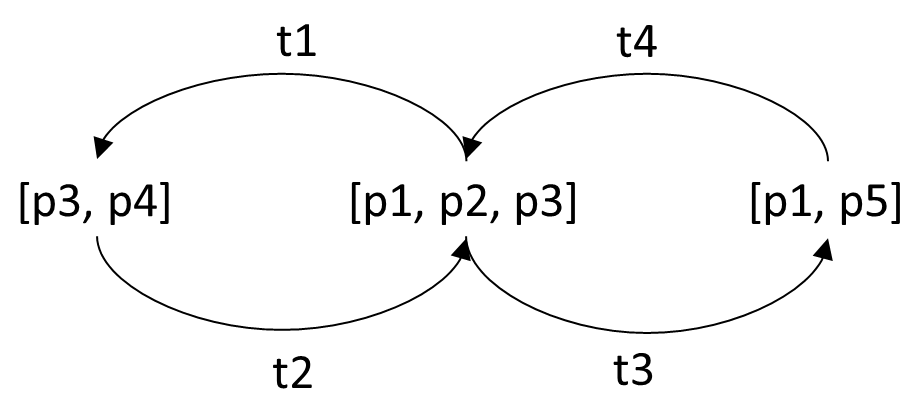
The concept of reachability graphs can be illustrated by means of mutual exclusion using the example of a single-tracked railway line, as represented in the following figure.

Petri Net (Right) for Single-Tracked Railway Line (Left) as Example for Mutual Exclusion



The transitions t1, t2, t3, and t4 are the actions of a train driving from north to south, an exiting train from north to south, a train driving from south to north, and an exiting train from south to north, respectively. Initially, places p1, p2,and p3 are each marked with one token, which enables both t1 and t3. We can represent the reachability graph for the Petri net on the basis of this initial marking as follows.

Reachability Graph for Petri Net



Using this representation we can directly obtain the set of reachable markings on the basis of the initial marking m0, which are m’={[p1,p2,p3],[p3,p4],[p1,p5]}. In this example, the important property of the system is that the places p4 and p5 are not set in the same marking and therefore it is not possible for trains to drive from north to south and vice versa at the same time. Of course, this model simplifies the scenario to a high degree but, as we want to keep it simple, it serves our purpose for now. For further details on reachability graphs, the interested reader is referred to the literature (Ye et al., 2003; van der Aalst, 2019).

To analyze some properties of Petri nets in more detail, it is convenient to describe them mathematically by means of vectors and matrices. Each marking m of a Petri net (P,T,F,m0) can be represented as a vector of length |P|, where each element of m contains the number of tokens ni∈ ℕ0 corresponding to the place pi∈P, thus

If we have a Petri net with six places, where p1, p2, and p4 contain one token at marking m, we can represent it by

Furthermore, a transition can be represented by a vector capturing the consumption and production of tokens in the places pi after firing

where w is the particular weight of the arrow (+1/-1 for condition-event nets). If we consider a transition t with ●t={p1,p2} and t●={p5} and a total number of six places in a condition-event Petri net, then we can write

This representation using vectors allows for a convenient calculation of the successive marking m2 after firing of t by means of vector addition:

By marking and transition we obtain

The whole set of transitions of a Petri net can be captured by means of a matrix N, where each column represents a single transition, i.e., N∈ ℕ0|P|×|T|*.* A net with three transitions and six places would yield

where the element is the action of transition tj on place pi.

Apart from a representation by means of one matrix, one can also separate it into two different matrices, where the post-incidence matrixPostcaptures flows from transitions to places and the pre-incidence matrixPre those from places to transitions. In that case, N=Post-Pre.

**Invariant**

This is a property of an object that does not change on the basis of certain manipulations.

In considering properties of Petri nets, **invariants** are of special interest. There are several possible properties that can be invariants, such as

* a combination of the number of tokens that does not change,
* a particular transition that cannot fire,
* a sequence of transitions that creates a loop, and
* a net that is deadlock free and thus can continue to change the marking.

A distinction is made between P-invariants (place invariants) and T-invariants (transition invariants) (Desel & Reisig, 1996).

P-invariants are characterized by the property that a linear combination of the number of place tokens ni is invariant with respect to fired transitions, i.e., it can be defined as a constant sum C of ni in the places pi∈P

where xi is the weight for the particular number of tokens of the i-th place. Each transition tj with weights xi has to satisfy

and thus for all transitions

For each solution x of the linear system of equations the weighted sum is invariant, while each net has the trivial invariant x=0, which is generally not of importance. Formally, we can define a P-invariant (place-invariant) as follows (David & Alla, 2010).

For a Petri net described by the matrix N=(P,T,F), I∈ ℤP is called place-invariant of N, if NT∙I=0, where 0∈ ℤTonly contains 0. ℤP and ℤT define all maps from P to ℤ and T to ℤ, respectively.

A Petri net is called “covered by place invariants” if there is a P-invariant that assigns positive values to all places (Starke, 1990).

In T-invariants, the execution of a sequence of transitions always leads back to the initial marking, while the order of transitions is arbitrary, as long as there are enough tokens in each preset, i.e., the transition is enabled. We can describe the change of tokens initiated by a sequence t of k transitions as a vector

where ni ∈ ℕ0 represents the number of occurrence of each transition ti in t and n is called the Parikh vector of t. Similar to the definition of place-invariants, we can define a linear system of equations that has to be satisfied for a certain Parikh vector. For each place pj∈P the equation

has to hold and thus

The T-invariant n=0 is a trivial solution for all Petri nets and generally not of importance. Formally, we can define T-invariants (transition-invariants) as follows (David & Alla, 2010).

For a Petri net described by the matrix N=(P,T,F), I∈ ℕ0T is called transition-invariant of N, if N∙I=0, where 0∈ ℤP only contains 0. ℕ0T and ℤP define all maps from T to ℕ0 and P to ℤ, respectively.

A Petri net is called “covered by transition invariants” if there is a T-invariant that assigns positive values to all transitions (Starke, 1990).

|  |
| --- |
| Summary |
| Petri nets are a convenient technique to model and analyze the behavior of discrete distributed systems. They consist of places, transitions, arcs, and tokens, which represent the states, actions, and the transitions between those and the shared resources of a system, whereas a transition always has to follow a place and vice versa. Furthermore, markings are the distributions of tokens over the places.  There are several variants of Petri nets. Condition-event nets limit the maximum number of tokens for each place to one. If a particular place is set with a token, it can be seen as a condition that is satisfied, which, in turn, leads to an event (the firing of a transition). Workflow nets have exactly one initial and one final place. Place-transition nets are extended by a weight function defining the weight of each arc. A transition becomes enabled and will be able to fire if and only if the numbers of tokens are set at the input places, while it consumes that same amount of tokens and produces as many new ones as defined by the particular weight in the output place. Colored Petri nets distinguish between multiple types of markers, which allows for more complex conditions when transitions can fire.  The reachability set of a Petri net gives insight into the possible markings that can be reached from a certain marking and are therefore a method for analysis. The reachability set of a marking is the set of all markings that can be reached from it, while the reachability set of a net is the set of all markings that can be reached from the initial marking.  Mathematically, Petri nets can be represented as matrices, which can be used to analyze their behavior. They can be invariant on the basis of certain properties. The term place-invariant refers to the property for which a linear combination of the number of place tokens is invariant with respect to fired transitions, while transition-invariant refers to the fact that the execution of a sequence of transitions always leads back to the initial marking. |

# Unit 4 – Timed Models

**Study Goals**

On completion of this unit, you will have learned …

… what types of timed models exist.

… how finite automata can be extended by a timing component to accept timed words.

… what the Markov property and a Markov process are.

… the basics of queuing theory.

… about different types of timed Petri nets.

# 4. Timed Models

## Introduction

Timed systems play an essential role in real-world applications. Therefore, modeling their behavior and their correctness is essential in order to ensure certain system properties. Can a residual current detector be triggered fast enough after inserting something in the power supply? Do the airbags deploy fast enough after an accident? These are only two examples of systems for which timing has to be modeled precisely. In this unit, we will take a closer look at some common timed models, such as timed automata, which extend the principle of finite automata by adding a set of real-value clocks; Markov processes, which are stochastic processes that can be used to determine probability for future events; queuing theory, which deals with queuing systems and the question of stability and waiting time; and timed Petri nets, which are extensions of the classic technique that models certain transitions to be timed. Regardless of the underlying timed model, fields of application range from computers and telecommunication systems to logistics and traffic systems.

## 4.1 Timed Automata

Let us recap briefly what finite automata are used for and what kinds of systems can be modeled by them. Finite automata consist of states, transitions, and actions and can be used as acceptors or transducers of regular languages. Timing is not modeled by the classic variant of this automaton, instead, state transitions are triggered by actions such as entering a certain character. The idea behind timed automata is to add a finite set of real-value clocks to the model so that the clock values will all increase with the same speed during a run of the timed automaton. The guards enable or disable transitions equipped with a time value that is compared to the current clock time, thereby regulating the system’s behavior (Alur & Dill, 1994).

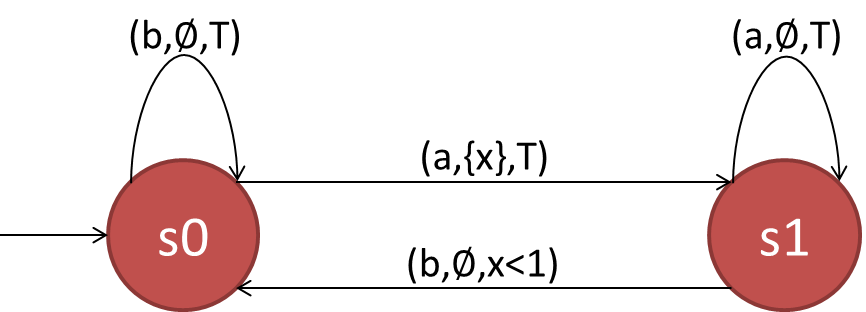
Let us have a look at an example to understand the basic concept of timed automata. To do so, we first need to define timed words and timed languages, which are extensions of words and formal languages (Alur & Dill, 1994).

Given an alphabet Σ, a timed word w can be defined by the sequence w=(σ0,t0) ( σ1,t1)… with σi∈Σ and ti∈ ℝ+, where ti≤ti+1.

A set of timed words L ⊆ Σ\* is called a timed language (over Σ).

For our example, we consider an alphabet Σ={a,b} and the language L over Σ, where an a is followed by a b in the next time unit T. The timed automaton accepting timed words w of L is shown in the following figure.

Acceptor for Timed Words w of L



The timed automaton stays in its initial state s0 as long as no character or b is entered before a. Entering a triggers a transition from s0 to s1 along with a clock x measuring the time that elapsed. If the first a is followed by a second a, the word is rejected within the run. The timed word is only accepted if a b is entered within less than one time unit, which is controlled by the guard x<1. By comparing the timed automaton with the classic one, we can see that their functionalities are quite similar, except for the additional comparison of the elapsed time. Formally, we can now define a timed automaton as follows (Alur & Dill, 1994).

A finite automaton is described by a tuple A={K,Σ,C,E,S,F}, where

K is a finite set of states,

Σ is an alphabet,

C is a finite set of clocks,

E is a set of transitions E⊆K×Σ×B(C)×P(C)×K, where B(C) and P(C) are the set of clock constraints and the powerset of C, respectively,

S⊆K is the set of initial states, and

F⊆K is a set of final states.

We call the transition from state s to s’ with character σ, guard g and clock reset r an edge e=(s,σ,g,r,s’)∈E.

**Clock valuation**

A function of the set of states to the set of non-negative real.

The tuple s=(s,γ) with state s and **clock valuation** γ is called the extended state.

The formal definition does not differ much from that of the classic automaton, with the only addition of the set of clocks and the change of the definition of transitions to incorporate timing. Furthermore, the introduction of the extended state s restricts the total determinism of a run because the clock valuation always has to be taken into consideration. Formally, a run of a timed automaton is defined as follows (Alur & Dill, 1994).

Given the timed automaton A={K,Σ,C,E,S,F} a run is defined by the sequence (s0,γ0)(s1,γ1), where

s0∈K,

∀i≥1∃e∈E with e=(si-1,σi,gi,ri,si) and

γi-1+ti-ti-1 satisfies gi,

γi=(γi-1+ti-ti-1), where the clocks are reset, thus ri→0.

Just as finite automata, timed automata can be distinguished between deterministic and nondeterministic; the determinism of a timed automaton can be defined as follows (Alur & Dill, 1994).

A timed automaton A={K,Σ,C,E,S,F} is deterministic if

S={s0},

for all pairs transitions e and e’ with e=(s,σ,g,r,s’) and e’=(s,σ,g’,r’,s’’) the sets of clock valuations γ and γ’ that satisfy g and g’, respectively, are disjoint, thus γ∩γ’=∅.

## 4.2 Markov Processes

**Stochastic process**

This is the mathematical description of random processes.

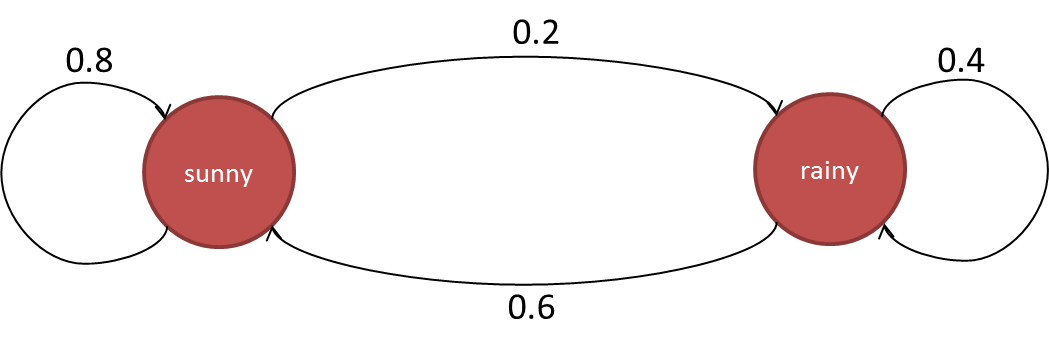
A Markov process is a **stochastic process** in which the future is independent from the past. While the term Markov chain is used synonymously in most cases, sometimes a differentiation between continuous (Markov process) and discrete processes (Markov chain) is made. The purpose of this is to determine probabilities for certain events in the future. Simply put, the current state of a system satisfying the Markov property captures all past states, i.e., the transitions only depend on the current state. Formally, this property can by defined as follows (Dynkin, 2012).

Given a set of random variables = (Xt)t∈ ℕ , where Xt takes values from the no more than countable set S={s1,s2,…},

is called Markov property. P(a|b) is the conditional probability of a given b. Y is called Markov chain or Markov process and it is called homogenous if the transition probabilities do not change.

The definition may seem a bit confusing at first. How can the current state contain all previous information and also be sufficient to predict future states? To illustrate that property, we can consider the following example. Suppose that a sunny day is followed by a second sunny day with an 80% probability. However, the probability of rain the day after a rainy day is 40%. This process can be illustrated by the following transition graph.

Transition Graph for Process of Weather



The first thing that we can observe is that the sum of probabilities with origin in the same state has to equal one, due to the fact that this is a stochastic process. In this way, it is ensured that all possible transitions are covered. The probabilities can be captured by means of the transition matrix

where the element pij gives the probability that weather j will follow a day with weather i. Given an initial stochastic vector x0, which represents the initial probability for that particular weather, we can predict the weather for the following days by multiplication with the transition matrix from the right, e.g., x1=x0∙P. Suppose we now know with a probability of 100% that today is a sunny day, the probability vector for tomorrow yields

The weather for the day after tomorrow yields

The general rule can be defined as xn=x0∙Pn=xn-1∙P and that corresponds exactly to the definition of the Markov property. Knowing the probability vector for the weather of the previous day is sufficient to predict the probabilities for all days to come. This also works the other way around, i.e., we can give a probability for past weather. In this case, the steady-state vector is particularly important to see what happens if we predict infinitely into the future. This can be represented by

and it is obvious that the uncertainty grows with each additional time step. The steady-state vector q can be obtained from P directly because it represents an eigenvector with eigenvalue 1 (Van Kampen, 1992). Let us calculate the steady-state weather for our example.

(1)

(2)

with I being the identity matrix

(3)

(4)

(5)

(6)

and we now that

(7)

Rearranging (7) and inserting it into (6) yields q1=0.75 and consequently q2=0.25. The property of a steady-state vector is that it does not change when multiplied by P. For our example

thus, the calculated values represent the average probabilities for that particular weather.

Let us now have a look at some of the attributes that characterize Markov processes. If all states can be reached from any other state by a sequence of transitions, the process is called irreducible (Rio, 2017). The following figure shows the transition graphs of a not irreducible (a) and an irreducible (b) Markov process, where the particular probabilities are omitted.

Transition Graph for Markov Process

## 

It is clear why a) is not irreducible: From state 4 none of the other states can be reached. In contrast, b) is irreducible because all states can be reached from each other state. A state of a Markov process is called periodic with period d—or d-periodic—if it returns to itself after at least d steps (Gebali, 2015). If d=1 then it is called aperiodic and if all states of the process are aperiodic then it is called an aperiodic Markov process. State 2 from the transition graph b) in the figure above is 2-periodic, whereas all other states are aperiodic.

Another important attribute is recurrence and, related to that, transience. A state is called recurrent if it is almost sure that it will be reached again after leaving it and transient if the probability to never reach it again is larger than zero. Consequently, a Markov process is called recurrent or transient if all of its states are recurrent or transient. A commonly used method for the determination of recurrence is the Green function (Sigelle et al., 2009). A state is called absorbing when it can never be left after it is reached, where the particular absorbing probability is of importance in most cases (Walter & Contreras, 2012). Reversible Markov processes are invariant with respect to time, i.e., it cannot be distinguished if they run back- or forwards (Stroock, 2013).

We want to keep the view of the attributes of Markov processes descriptive without going into detail about the underlying mathematical concepts. For further details on this topic, the interested reader is referred to the literature.

## 4.3 Queuing Theory

Waiting lines, also known as queues, are omnipresent in our daily life and play an important role in various different processes. To describe such situations, the queuing theory was formulated as a mathematical study of waiting lines. A study of the Danish Telephone Company for the determination of the optimal number of telephone lines in 1909 is regarded as the first work in this field (Erlang, 1909; Swamidass, 2000). Generally, a queuing process consists of two parts: a customer requesting a certain service and a server delivering said service. Consider a waiting line at a coffee shop. The customer requests a coffee, while the barista serves it. Of course, he cannot serve all orders at once and thus each customer has to wait for their turn. Another example would be a network printer that gets print requests from multiple people. In this case, the printer and the people would take on the role of server and customers, respectively.

In the theory of queuing the whole system of such waiting lines is investigated, including aspects such as customer arrival rate, the number of customers and servers, or the properties of the waiting area (Cassandras & Lafortune, 2009). A queue—or queuing node—can be regarded as a black box, where the customers arrive, wait for a certain amount of time for their requests to be processed, and depart afterwards. The black box assumption is used to say that we do not really know what happens inside, which is of course not completely correct because there needs to be some known information about it. A queuing node is generally classified on the basis of the Kendall notation using the notation A/S/c/K/N/D (Kendall, 1953), where the symbols are defined as follows.

* A: arrival process
* S: mathematical distribution of service time
* c: number of servers
* K: capacity of queue (maximum number of customers)
* N: size of customer population
* D: queuing discipline

The symbols K and N are omitted if they are unlimited. If no value for D is given, a first-in-first-out (FIFO) order is assumed as the default. A queuing process, where the customer departs immediately after their order has been processed, is called a queue with no buffer or no waiting area. If there is a waiting area of size n it is referred to as a queue with buffer of size n.

Let us return to our previous example of a coffee shop to explain the different components of a single queuing node. There are certain properties that we can identify immediately. We know that the number of customers that can be served at a time is one (number of servers). Furthermore, the first customer that arrives is also the first one to depart, making this a FIFO order (queuing discipline D). The maximum number of customers (K) and the size of customer population (N) are unlimited. For the arrival process (A) and the mathematical distribution of service time (S) a certain process or distribution has to be defined. For instance, in the example we chose, A and S are generally set to be a **Poisson process** and an exponential distribution, respectively (Lipsky, 2009). Using Kendall’s notation, those systems are called **M/M/1 queues**, where M stands for A and S is the code for Markovian. For a complete overview of the different codes of the notation, the interested reader is referred to the literature. The behavior of the single queuing node of our example represents a birth-death-process because the new customers that arrive and those that leave can be considered as new objects that are born and die in the system. This can be illustrated as shown in the following figure.

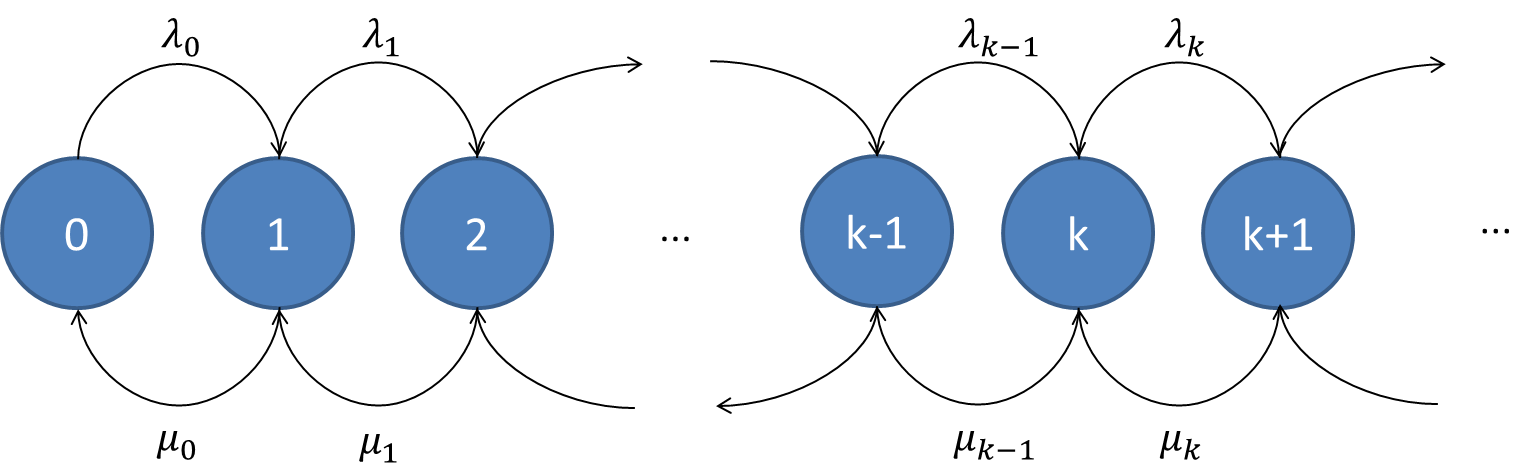
**Poisson process**

This is a renewal process where the renewals are Poisson-distributed.

**M/M/1 queues**

This is the most elementary type of queue with one server. An extension to a number of c servers is the M/M/c queue.

Queuing Node as Birth-Death-Process



The states of the process are represented by the circles, where k is the number of customers and λi, μi are the birth and death rates, respectively, which are generally considered as constant, i.e., λ= λ0, λ1,…,λk and μ=μ0,μ1,…,μk. Often, it is of interest to determine a steady state of such a birth-death-process, which can be done by means of the balance equations (Harrison & Patel, 1992). With Pn being the steady state probability of the system to be in the state n, the equations are given by

Methods of queuing theory can be used to describe behaviors and properties of queues, such as the average waiting time, or to optimize those in order to obtain the most efficient systems. A basic theorem of this theory is Little’s law, defined by John Little (Little & Graves, 2008), which represents a relation between the average number of customers L, the average time spent W, and the average arrival rate λ into a stationary system. As an equation it can be formulated as

and it holds without knowing any other information about the particular queue. Let us investigate what Little’s law means for our coffee shop example, where we assume the average number of customers to be 12 and that the barista can serve three customers per minute. The average waiting time for a coffee can be calculated by

There are various fields of application where queuing theory plays a significant role, such as logistics (Żak & Jacyna-Gołda, 2013; Masek et al., 2015), computing (Vilaplana et al., 2014), industrial engineering (Marsudi & Shafeek, 2014; Rashid et al., 2015), telecommunications (Giambene, 2005), or healthcare (Mehandiratta, 2011; Fomundam & Herrmann, 2007). For further details on this interesting topic, the interested reader is referred to the literature.

## 4.4 Timed Petri Nets

As previously discussed, Petri nets can be used to model and analyze the behavior of discrete and distributed systems. In some situations, the classic approach of modeling only on the basis of the arrangement of places and transitions and the flows between them is not sufficient for an appropriate system representation. In classic Petri nets, all transitions can fire at the very moment they become enabled, which does not correspond to reality for many systems. One approach to deal with that issue is the extension by firing pre-conditions where a time delay is added to the transitions, which means that the pre-conditions have to hold before completing the firing (Pawlewski, 2012). Petri nets that are augmented by a time delay are commonly known as timed Petri nets. The time specification can also influence the other parts of a net, i.e., places, arcs, and tokens, and it can be deterministic or stochastic (Zuberek, 1991). Let us first have a look at deterministic timed transitions Petri nets, which were the first approach used to assign time labels to each transition (Ramchandani, 1974). Formally, those can be defined as follows (Merlin, 1974).

A deterministic timed transitions Petri net is defined as a 4-tuple (P,T,F,τ), where

P is a finite set of places,

T is a finite set of transitions, where T∩P,

F is a set of flow relations F⊆(P×T)∪(T×P), also called arcs, and

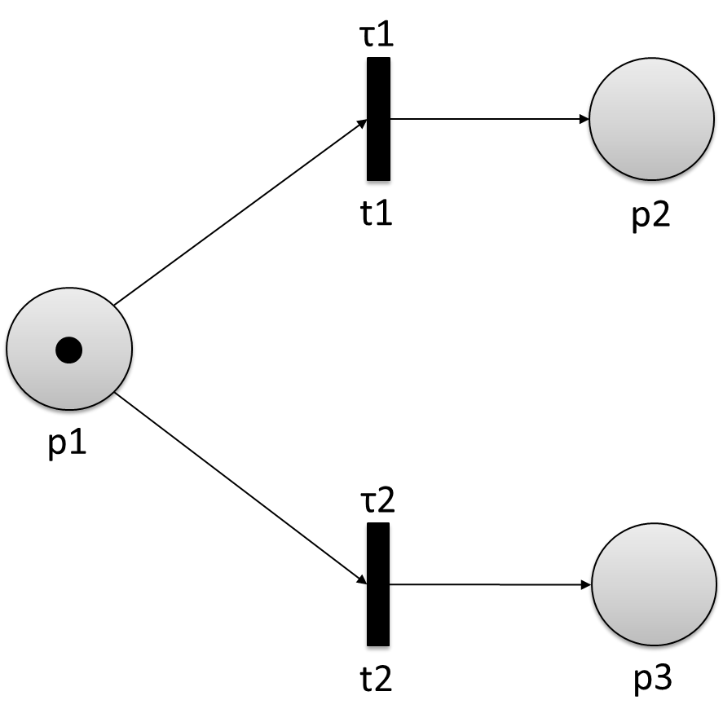
τ is a time function associating a time delay to each transition τ: T→ℝ+.

In comparison to classic Petri nets—where transitions can fire immediately—the firing pre-condition of a certain transition ti∈T in DTTPNs is defined as follows.

* The preset ●p of ti is marked continuously with the required tokens specified by the connecting arcs for the time interval [τ-τi,τ] with the associated time delay τi.
* The postset p● of ti is marked with the tokens specified by the connecting arcs at a time τ.

Let us consider the following deterministic timed transitions Petri net to illustrate the concept.

Example of Deterministic Timed Transitions Petri Net



What happens if τ1<τ2? The transition t1 can only fire if p1 is marked for the time interval [τ-τ1,τ] and produces a token in p2 at τ. Transition t2, however, fires if p1 is marked in the time interval [τ-τ2,τ]. Hence, when the time delay τ1 is reached, t1 fires and consumes the marking from p1. Since now p1 is no longer marked in the time interval, the firing pre-condition of t2 does not hold anymore and therefore it will not fire, although p1 was marked in the interval [τ-τ1,τ].

The same augmentation presented for the transitions of a Petri net can be applied to the places and arcs in order to build deterministic timed places Petri nets (DTPPN) and deterministic timed arcs Petri nets (DTAPN), respectively (Zuberek, 1991).

Another variant of timing specification is defined by the time Petri nets (TPN), where each transition t1 is associated with two time values αi and βi representing the minimum and maximum waiting time before firing after being enabled (Merlin, 1974). Thus, a transition that was enabled at a time τ can only fire in the time interval [τ+αi,τ+βi]. Formally, it can be defined as follows.

A time Petri net is defined as the tuple (P,T,F,IS), where

(P,T,F) is a Petri net and

IS is the static interval function t→ IS(t)⊆ ℝ+ defining rational time bounds.

A further extension of Petri nets that adds a nondeterministic time delay to each transition is the stochastic Petri net, which can be defined as follows (Marsan, 1988).

A stochastic Petri net is defined as the tuple (P,T,F,Λ), where

(P,T,F) is a Petri net and

Λ is a function assigning a firing rate λi, represented by a random variable, to each transition.

Reachability graphs can be created for timed Petri nets just as they are for their classic versions. For stochastic Petri nets this results in an interesting property: Their reachability graph can be mapped directly onto a Markov process due to the fact that their states only depend on the current one (Marsan, 1988). In the map, the states and transition firings with firing rate λi of the graph are represented as Markov process states and Markov state transitions with probability λi, respectively.

|  |
| --- |
| Summary |
| Finite automata can be extended by a finite set of real-valued clocks that increase their values all with the same speed during a run. In this way, timed words—formal words where the characters are augmented by timings—can be accepted. The guards enable or disable transitions that are equipped with time values by comparing those to the current clock values.  The Markov property refers to the property of a system in which the current state contains all information about the past, which means that the transition probabilities only depend on the current state. Markov processes are based on this assumption and serve the purpose of determining probabilities for future events of a system. They can have certain attributes such as periodicity, irreducibility, recurrence, or transience.  Queuing theory is the study of the mathematical description of waiting lines. A queuing process can be classified on the basis of Kendall’s notation containing the arrival process, the mathematical distribution of service time, the number of servers, the capacity of the queue, the size of the customer population, and the queuing discipline. Many queuing nodes are called birth-death-processes, meaning that a customer that arrives leaves right after being served. Little’s law gives a direct relation between the average number of customers, the average time spent at a specific location, and the average arrival rate into a stationary system. The fields of applications of queuing theory are very diverse, e.g., healthcare, logistics, industrial engineering, or telecommunications.  Different models exist to extend Petri nets by a timing aspect, which can generally be divided into deterministic ad stochastic. While the timings are known for the former, they represent random variables for the latter. Deterministic timed transitions Petri nets, deterministic timed places Petri nets, and deterministic timed arcs Petri nets specify timing for each transition, place, and arc, respectively. Time Petri nets use static interval functions to define rational time bounds, while stochastic Petri nets assign random firing rates to each transition to obtain a nondeterministic behavior. |

# Unit 5 – Simulation of Discrete Event Systems

#### Study Goals

On completion of this unit, you will have learned …

… how discrete event and continuous systems can be differentiated from each other.

… what the main components of discrete event systems are.

… how the simulation of discrete event systems works.

… how events are scheduled in the simulation of discrete event systems.

… what kinds of performance measures can be used for analysis.

… to create basic simulations of discrete event systems using SimPy.

# 5. Simulation of Discrete Event Systems

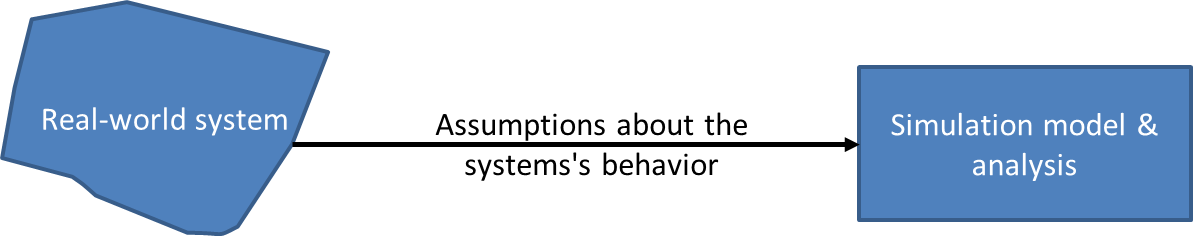
## Introduction

In order to analyze systems, it is necessary to create abstract models which can be used for simulations under certain mathematical and logical assumptions. Discrete event systems represent one class of systems whose state does not change continuously but only at the occurrence of events. In this unit, we will have a look at the basic concepts of discrete event systems and their simulation. Firstly, we will present the main components of these systems and show the differences with respect to continuous systems. Then, we will explore the basic steps of creating simulations and the working principle of discrete event system simulations, with a focus on the event-scheduling/time-advance algorithm. We will also cover some basic aspects of the performance analysis of discrete event systems and touch upon different performance measures. Finally, we will take a closer look at existing simulation software and show the development of a very simple system simulation using SimPy.

## 5.1 Basic Concepts

In general, simulations imitate the behavior and operation of real-word systems over time by developing relationships between particular entities. This can be used, for example, to analyze the performance of a new system or to predict the impact of changes that were made to existing ones. Complex systems can thus be studied to investigate possible improvements by analyzing the simulation model. The general relationship between real-world systems and simulation models is shown in the following figure.

Relationship between Real-World System and Simulation Model



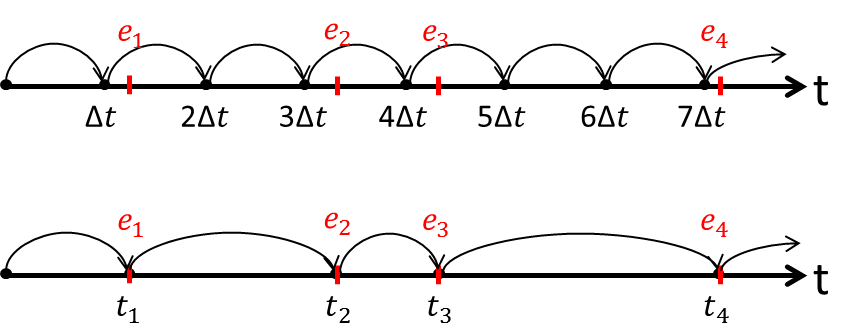
A system can be either continuous or discrete. The state variables of continuous systems change continuously over time, while those of discrete systems only change at a discrete set of points in time (Banks, 2005). Most natural processes and physical systems are continuous, such as the decay of radioactive elements, the flow of water over a dam, or the amount of alcohol that is produced during fermentation. Instead, typical examples for discrete systems are queues or **discrete manufacturing**. While continuous simulation techniques use differential equations to describe the changes of the state’s variables over time, the system’s operations are modeled as discrete sequences of events in discrete-event simulations. A discrete event must not be confused with a discrete-event simulation: The former relies upon systems that change based on events, while the latter considers phenomena that are countable, such as the number of animals that passed through a gate.

**Discrete manufacturing**

This refers to the production of distinct products that can be counted.

In this unit, we will only consider discrete-event simulation, which can generally be classified into next-event time progression and fixed-increment time progression approaches (Bachelet, 1998). Next-event time progression describes the approach where the simulation time jumps to the time when the next event occurs under the assumption that the system’s state between two events is constant. Fixed-increment time progression follows a different approach and splits the time up into discrete time steps that define the points where the system’s state is updated (Bachelet, 1998). Due to the known points of state changes and the associated reduction of simulation steps, next-event time progression allows for a faster simulation than fixed-increment time progression. Let us have a look at the following figure to get a deeper understanding of the difference between the two approaches.

Fixed-Increment Time Progression (Top) and Next-Event Time Progression (Bottom)



The top section of the figure shows the principle of fixed-increment time progression with time slices of size Δt and events e1, e2, e3, and e4. It is immediately evident that there are many unnecessary simulation steps that do not show any changes in the system’s state. Furthermore, an additional disadvantage of this approach becomes clear: If the time steps chosen are too large and two events lie close together, the change of the first event is skipped. The next-event time progression shown at the bottom of the figure is a much more convenient approach because it needs fewer simulation steps and can capture all events.

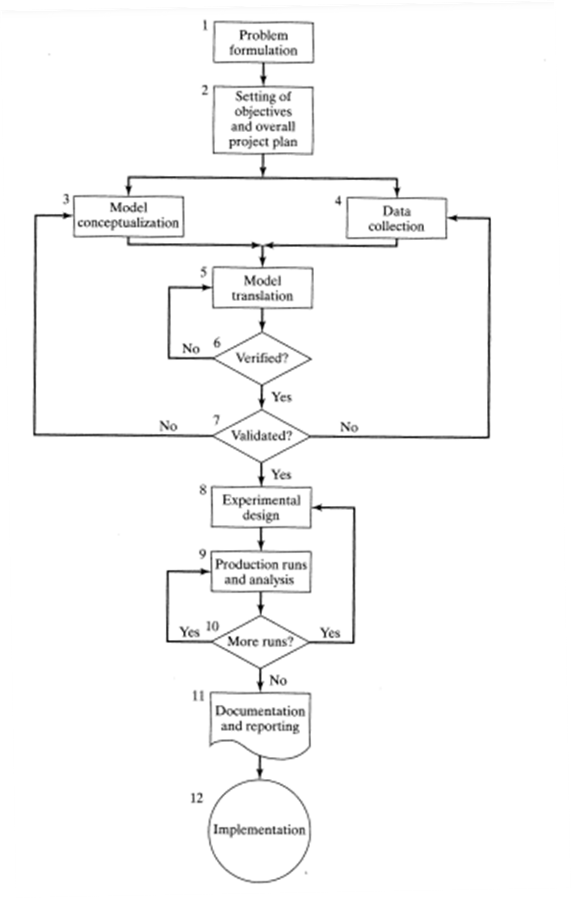
The basic concepts of discrete-event simulation are defined in the table below (Banks, 2005).

Basic Concepts of Discrete-Event Simulation

|  |  |
| --- | --- |
| System | A list of interacting entities that execute operations |
| Model | An abstract description of a system incorporating mathematical and logical relationships |
| System state | A collection of variables that describe the system |
| Entity | An object in the system whose behavior is represented in the model |
| Attributes | These capture the properties of the different entities. |
| List | A collection of entities in a logical order, such as first-in-first-out (FIFO) |
| Event | These lead to a change in the system. |
| Event notice | A record of the time of occurrence for an event and its type, i.e., the operation that is executed when the event occurs |
| List of events | The events of the system (also known as pending event set or future event list) are captured by a list of events. |
| Activity | Specified duration of time with known starting point. It can be specified deterministically, statistically, or as a function depending on a system’s variable. |
| Delay | Unspecified duration of time with unknown end |
| Clock | The simulation time is controlled and captured by a clock that either generates constant time steps in the case of fixed-increment time progression or variable ones in the case of next-event time progression. |

The general development process of a simulation model for a certain system is illustrated in the block diagram below (Banks, 2005).

Steps in a Simulation Study



The process starts with the problem formulation (1), where the person in charge analyzes and formulates the problem, and which is followed by the setting of objectives and the overall project plan (2). Afterwards, a loop consisting of model conceptualization (3), data collection (4), model translation (5), verification (6), and validation (7) follows. This means that an abstract model of the system is created incrementally by adapting initial assumptions of behavior, which leads to a useful approximation. Simultaneously, appropriate data is collected from the system and the abstract model is translated to a simulation model that is verified, i.e., checked to see whether it provides a correct functionality, and validated—checked to ensure that it appropriately represents the real system. After final validation, another loop consisting of experimental design (8) and production runs and analysis (9) starts, where the parameters of the experiment, such as the number of simulation runs or the number of replications, are set and a performance analysis is carried out. Finally, the documentation and reporting is done (11) and the final simulation model implemented (12). The interested reader is referred to the literature on the design process of simulation models (Banks, 2005).

## 5.2 Working Principles

To fully understand the working principles of discrete event systems and their simulation, it is important to take a closer look at the differences between activities and delays. Activities are called unconditional waits because both the start and end point are known (even if they may be defined stochastically), whereas delays are conditional waits because their end depends on a system’s condition (Banks, 2005). Since the end of an activity is known, it thus represents an event—called primary event—which is placed on the future event list. Instead, the entity associated to a delay is put on a “waiting” list and its completion is called secondary event. The table below summarizes the differences between activities and delays.

Activities vs. Delays

|  |  |
| --- | --- |
| **Activity** | **Delay** |
| Unconditional wait | Conditional wait |
| Event (end of activity) placed on future event list | Entity placed on a “waiting” list |
| Primary event | Secondary event |

A discrete-event simulation generates a snapshot of the system for each simulation time step, which consists of the system’s state, the status of entities and **sets**, the existing lists (at least the future event list with activities in progress and, if available, the “waiting” list), the values of the cumulative statistics for summary statistics (means, variances, etc.) at the end of the simulation, and the clock (Banks, 2005). The following table shows an example of such a snapshot for time t.

**Sets**

In a discrete-event simulation, sets capture the required information for the calculation of performance metrics.

Example of Snapshot of System for Simulation Time t

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Clock** | **System state** | **Entities and attributes** | **Set 1** | **Set 2** | **…** | **Future event list** | **Cumulative statistics** |
| t | (x,y,z,…) |  |  |  |  | (3, t1) – Type 3 event occurs at t1  (1, t2) – Type 1 event occurs at t2 |  |

As seen in the table above, not all components have to be included in a snapshot. Now, let us take a look at the mechanism for advancing the simulation time and for scheduling the events, also known as event-scheduling/time-advance algorithm.

As the name suggests, the aim of this algorithm is to advance the simulation time and to schedule events in order to guarantee that each event happens at the correct time. This is based on the future event list that contains all of the scheduled event notices in chronological order, i.e., t<t1≤ t2≤…tn, where t is the current value of the simulation time and t1 represents the imminent event (Banks, 2005). We already mentioned that activities cause a new end-activity event to be placed into the future event list together with the event time. The latter can be calculated by means of the given duration or by drawing a sample from the given statistical distribution in the case of deterministic and stochastic specification, respectively. It is clear that the future event list changes over time and the simulation performance depends on the management of it. There are five major steps that are needed for event scheduling (Banks, 2005).

1. The event notice for the imminent event is removed from the future event list.
2. The clock is advanced to the event time of the removed event.
3. The imminent event is executed, which results in an update of the system’s state and a change in the entity’s attributes and set memberships.
4. Event notices of new future events are placed into the future event list.
5. The statistics and counters of the simulation are updated.

Let us have a look at an example to see how the procedure works in practice. A good process to be simulated as a discrete event system is a street with traffic lights and a pedestrian crossing, where we reduce the complexity by only considering green and red lights. In our assumption, the traffic lights for the vehicles are green until a pedestrian arrives and presses the button to cross the street. At that moment, the lights for the vehicles and for the pedestrians turn red and green, respectively, before reverting back to their initial state after a specified time. For the model of this system we can derive four different states.

* State 1: green lights on the street and red lights at the crossing
* State 2: red lights on the street and at the crossing; in the imminent state, the lights will turn green at the crossing
* State 3: red lights on the street and green lights at the crossing
* State 4: red lights on the street and at the crossing; in the imminent state, the lights will turn green on the street

The transitions between the states are represented in the following figure.

State Transitions for Example of Street with Pedestrian Crossing

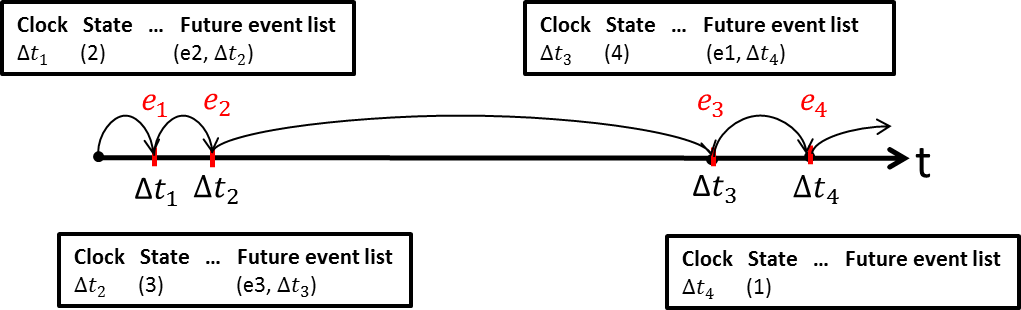


When a pedestrian presses the button, the system switches to state 2, where it stays for a certain time t1; both lights remain red before switching to state 3, where the pedestrian light turns green. After another specified time t2, the system switches to state 4; both lights remain red before switching to the initial state 1 after a certain time t3. We now want to have a look at the event scheduling for this system model and start at time t=0 with the initial conditions. We set our times to be t1=2, t2=10, and t3=2. No events have been placed into the future event list yet because no activity has been started. Now, a pedestrian arrives at t=20 and presses the button. This is called an **exogenous event** and results in a new event notice e1 to be placed into the future event list with the event time t=20+2. At simulation time t=22, the previously generated event e1 is removed from the list and executed, which leads to a change in the system’s state from state 2 to state 3. A new event notice e2 with event time t=22+10=32 is placed into the future event list, which represents the transition from state 3 to state 4. At time t=32, e2 is removed from the list, executed, and a new event notice e3 with event time t=32+2=34 is placed into the list representing the state transition from state 4 to state 1. Finally, at t=34, e3 is removed from the list and executed. The following figure shows the snapshots of the system for the simulation run.

**Exogenous event**

Thisis an event originating externally. The word “exogenous” comes from the Greek words ἔξω (éxō) and γένεια (géneia), meaning outside and to produce, respectively.

Snapshots of System for Simulation Steps



This basic simulation model could now be used to optimize the system, e.g., under the assumption of certain probability distribution for the arrival of pedestrians, one could simulate how the different times for the state transitions have to be adapted to allow for the crossing of a maximum number of pedestrians while keeping the amount of red phases for vehicles as low as possible.

## Self-Check Questions

1. Which of the following terms defines a delay?

* *conditional wait*
* unconditional wait
* event list

## 5.3 Performance Analysis

Output analysis is an important topic in the field of discrete-event system simulation and describes the process of examining the output data of a simulation to either evaluate the performance of a single system or to compare the performances of multiple system designs with each other (Banks, 2005). The performance of a system can be represented by a parameter θ and a simulation of a system model yields an estimator of θ with a precision either measured by the standard error or a confidence interval. By means of statistical performance analysis we now try to estimate that very precision and/or to determine the number of required observations to obtain a precision of a specified size (Banks, 2005). Two main issues have to be considered when analyzing a system, namely autocorrelation—possible statistical independence of events—and the initial conditions of the system, which might have a significant impact on the performance. In general, we differentiate between two types of simulations used for performance analysis, which are shortly described below (Banks, 2005).

#### Terminating/transient simulation

A terminating simulation starts at time t=0 under specified initial conditions and terminates at a certain event E (or set of events) with event time tE (or set of times). Some examples are, for instance, all shops with defined opening and closing times where the event E is the closing of the shop. Initial conditions are the number of servers and the absence of customers in the shop at t=0. The event time tE may not be known initially and a number of n simulation runs may yield the different values

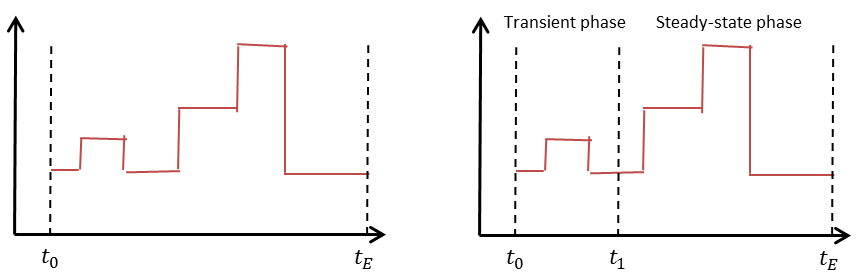
.

One of the purposes of this analysis can be to estimate the real event time tE.

#### Steady-state simulation

In comparison to terminating simulations, steady-state simulations run for a long time or even continuously. This simulation starts at time t=0 with initial conditions and runs for a time tE; both times are defined by the analyst. The previous example of a traffic light system would be a good representation of this type of simulation. The goal of the analysis is to obtain information about the steady-state of the system, i.e., to prove that the initial condition did not have an impact on it. The following figure shows the difference between the simulation types.

Transient Simulation (Left) and Steady-State Simulation (Right)



The simulation on the left shows the concept of transient simulation, which runs in the time interval from t0 to tE. On the right side is shown, instead, a steady-state simulation which reaches a steady-state phase (from t1 to tE) after a transient phase (t0 to t1). The type of simulation is not only defined by the system of interest itself but also by the objective of the analysis, e.g., a shop could be analyzed using transient simulation to investigate the behavior of a day or using steady-state simulation when trying to identify the properties for a longer period of time, for example, a year. Given that random variables—such as arrival rates or service times with certain distributions—are included in models of systems, the output will also consist of stochastic data. To estimate a performance parameter θ of a system after n simulation runs that yield the discrete time data {Y1,Y2,…,Yn} we can use different measures. The simplest one is the point estimator, which is defined as

**Bias**

This refers to the presence of a constant value that is superimposed to a signal.

and that is unbiased if

and biased if

with bias

,

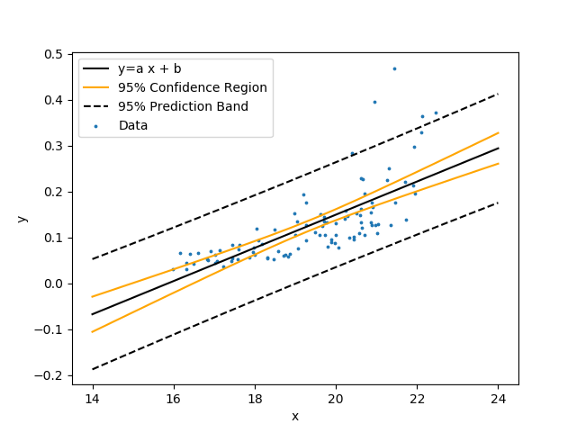
where E() is the expected value. While the former is the desired property, the latter represents the reality (Banks, 2005). Instead of the mean, the median (the 0.5 quantile of the data) is often used as performance measure, i.e., the data are split up into two parts of equal size, one containing the values smaller and the other the values larger than the median.

Besides the point estimator, the confidence interval estimation represents a widely used concept for performance analysis; in order to fully grasp this concept, we first have to understand the difference between measure of error and measure of risk. For the mean

and the variance

of a given sequence of samples X={X1,X2,…Xn} drawn from a normal distribution, the confidence interval gives us a region in which the performance parameter is included with a certain probability, i.e., where the error bound falls between and θ. The prediction interval, however, gives a region in which the future samples fall with a certain probability. In the following figure the difference between both measures can be observed.

Confidence and Prediction Interval for Data



For more details on this topic, the interested reader is referred to the literature (Banks, 2005).

## 5.4 Software Tools

After presenting the basics of discrete-event system simulation, we now want to show how the theory can be used in practice. There are many different kinds of simulation software available on the market and each has its own advantages and disadvantages. Commercial tools have been used in the industry and in academia for a long time. The high prices, ranging from several hundreds of euros up to thousands, are one of the main reasons why many small and medium-sized companies have renounced the implementation of appropriate simulation software for their systems (Dagkakis & Heavey, 2016). Within the last decades, it was mainly academics who promoted the development of open-source simulation software in order to overcome this limitation. A review done by Dagkakis and Heavey shows that most of the available open-source simulation tools use Java and C++ as programming languages, a characteristic that may be traced back to the higher speed of their performance if compared to other languages such as Python, C#, or Javascript (Dagkakis & Heavey, 2016). Python, however, has the advantage that it can be learned much more quickly than most other languages, which increases its attractiveness for users that are not programming experts. As of 2018, the most common commercial simulation tools are FlexSim, AnyLogic, Arena, and Emulate3D (Descreye Solutions, 2018), while the most common open-source tools are OMNeT++, NS-3, and SimPy (Dagkakis & Heavey, 2016). For more detailed information on these tools, the interested reader is referred to the literature (Dagkakis & Heavey, 2016; Descreye Solutions, 2018).

Let us now examine the modeling of a simple system using SimPy, which can be directly integrated into the Python environment. As for all other packages, the framework can be installed by means of

1. pip install simpy

In SimPy, the events of a system are called processes and they are handled inside environments. We are now able to create the model of our previous example of a street with pedestrian crossing and traffic lights with just a few lines of code.

1. **import** simpy
3. **def** TrafficLight(env):
4. **while** True:
5. **print** ("Vehicles: red, pedestrians: red at " + str(env.now))
6. # Both lights are red for 2 seconds
7. **yield** env.timeout(2)
9. **print** ("Vehicles: red, pedestrians: green at " + str(env.now))
10. # Pedestrian lights are green for 10 seconds
11. **yield** env.timeout(10)
13. **print** ("Vehicles: red, pedestrians: red at " + str(env.now))
14. # Both lights are red for 2 seconds
15. **yield** env.timeout(2)
17. **print** ("Vehicles: green, pedestrians: red at " + str(env.now))
18. # Vehicle lights are green for 10 seconds
19. **yield** env.timeout(10)
21. # Defining the environment variable
22. env = simpy.Environment()
24. # Add process to the environment
25. env.process(Traffic\_Light(env))
27. # Run process for 15 seconds
28. env.run(until=15)

In this model, the event of a pedestrian’s arrival is set to occur 10 seconds after the last cycle, which is done to keep the demonstration code as simple as possible. The output for a simulation time of 15 seconds is the following.

Vehicles: red, pedestrians: red **at** 0

Vehicles: red, pedestrians: green **at** 2

Vehicles: red, pedestrians: red **at** 12

Vehicles: green, pedestrians: red **at** 14

It must be said that this example is quite unspectacular and does not accurately represent reality. We can now introduce stochastic behavior in our model by generating a pedestrian arriving at a random time. Thus, we reformulate our scenario by creating a class Traffic containing two processes, one for the pedestrian generation and one for the traffic light sequence. Both processes are controlled by means of the events light\_cycle\_completed and random\_pedestrian\_stopping, which represent the termination of trafficLight and randomPedestrian, respectively, and simultaneously initiate the start of the other process. The following code shows the implementation of the modified scenario.

1. **import** simpy
2. **import** random
4. # Class containing traffic scenario
5. **class** Traffic(object):
6. **def** \_\_init\_\_(self, env):
7. # Defining the environment variable
8. self.env=env
9. # Define two processes
10. self.trafficLight\_proc=self.env.process(self.trafficLight())
11. self.pedestrian\_proc=self.env.process(self.randomPedestrian())
12. # Define two events
13. self.light\_cycle\_completed=self.env.event()
14. self.random\_pedestrian\_stopping=self.env.event()
15. # Process for generation of random pedestrian
16. **def** randomPedestrian(self):
17. **while** True:
18. # Generation of random time
19. t = random.expovariate(0.2)
20. **yield** self.env.timeout(t)
21. # Finish this process by triggering event
22. self.random\_pedestrian\_stopping.succeed()
23. self.random\_pedestrian\_stopping=self.env.event()
24. **yield** self.light\_cycle\_completed
25. # Process for traffic light
26. **def** trafficLight(self):
27. **while** True:
28. # Wait for stop of pedestrian generation
29. **yield** self.random\_pedestrian\_stopping
30. **print** ("Pedestrian arrives and presses button.")
31. **print** ("Vehicles: red, pedestrians: red at " + str(env.now))
32. # Both lights are red for 2 seconds
33. **yield** env.timeout(2)
34. **print** ("Vehicles: red, pedestrians: green at " + str(env.now))
35. # Pedestrian lights are green for 10 seconds
36. **yield** env.timeout(10)
37. **print** ("Vehicles: red, pedestrians: red at " + str(env.now))
38. # Both lights are red for 2 seconds
39. **yield** env.timeout(2)
40. **print** ("Vehicles: green, pedestrians: red at " + str(env.now))
41. # Finish this process by triggering event
42. self.light\_cycle\_completed.succeed()
43. self.light\_cycle\_completed=self.env.event()
45. env = simpy.Environment()
46. traffic=Traffic(env)
47. env.run(100)

The important parts of the program are lines 24 and 29, which represent the wait for an event to happen, and lines 22 and 42, which trigger the events. It has to be considered that, once an event has been triggered, it cannot be reused and must be redefined (lines 23 and 43). The following output shows the first three traffic light sequences as a result of the simulation.

Pedestrian arrives and presses button.

Vehicles: red, pedestrians: red **at** 7.719680824358208

Vehicles: red, pedestrians: green **at** 9.719680824358207

Vehicles: red, pedestrians: red **at** 19.719680824358207

Vehicles: green, pedestrians: red **at** 21.719680824358207

Pedestrian arrives and presses button.

Vehicles: red, pedestrians: red **at** 22.301853413396262

Vehicles: red, pedestrians: green **at** 24.301853413396262

Vehicles: red, pedestrians: red **at** 34.30185341339626

Vehicles: green, pedestrians: red **at** 36.30185341339626

Pedestrian arrives and presses button.

Vehicles: red, pedestrians: red **at** 43.72174233817722

Vehicles: red, pedestrians: green **at** 45.72174233817722

Vehicles: red, pedestrians: red **at** 55.72174233817722

Vehicles: green, pedestrians: red **at** 57.72174233817722

It can be seen that pedestrians arrive at random times and initiate a traffic light cycle to cross the street. Once the cycle is finished the process pauses and waits for a new event to happen. This behavior is already much more realistic compared to the first example and it is easy to extend this scenario to incorporate queues of pedestrians and cars. For a detailed study of this tool, the interested reader is referred to the literature (Matloff, 2008).

|  |
| --- |
| Summary |
| The simulation of discrete event systems is an important process for the analysis of real-world systems. Abstract models are derived based on mathematical and logical assumptions that capture reality as accurately as possible. In general, simulations can be distinguished into continuous, discrete, and discrete-event based. Continuous simulation calculates the system’s states at each time interval using differential equations; discrete simulation relies on phenomena that are countable; discrete-event simulation only updates the state parameters when certain defined events occur. This can be done using fixed-increment time progression or next-event time progression. While the former means that the simulation is updated at predefined time intervals, the latter uses events to trigger the calculation of new values.  Other important events that can occur and that have to be differentiated are activities and delays. Activities are called unconditional waits because their start and end point is known, while delays are conditional waits because their end depends on a system’s condition. At each simulation step, a snapshot of the system is generated which consists of the system’s state, the status of entities and sets, the existing lists, the values of the cumulative statistics for summary statistics at the end of the simulation, and the clock. The event-scheduling/time-advance algorithm is used for time advancing and scheduling of the events in a simulation. When analyzing performance, a distinction must be made between transient simulation and steady-state simulation, because the latter, after an initial phase, has another phase in which the behavior is assumed to be in a steady-state. The point-estimator and the confidence interval estimator are two commonly used methods for performance analysis.  There are several software tools for the simulation of discrete event systems, both commercial and open-source. SimPy represents a Python framework that can easily be used for modeling and simulation. |

# Unit 6 – Supervisory Control

#### Study Goals

On completion of this unit, you will have learned …

… the formal definition of supervisory control.

… what the main components of supervisory control are.

… what specifications for supervisors are.

… how supervisors can be synthesized.

… how the performance of supervisory controllers can be analyzed.

… about the different implementations of supervisory control.

# 6. Supervisory Control

## Introduction

Supervisory control is used to control multiple controllers within a system in an organized manner and by focusing on the overall behavior. It can be found in many real-world plants because the simplification of the control is tremendous. Nevertheless, the derivation of appropriate supervisors can be a difficult task for complicated systems and requires the formulation of specifications that define the behavior. In this unit, we will show how specifications are defined and how they are used. We will present the process of supervisory control synthesis based on the formulation of the controller synthesis problem and give an overview of existing methods for performance analysis, ending the unit with the implementation of supervisors.

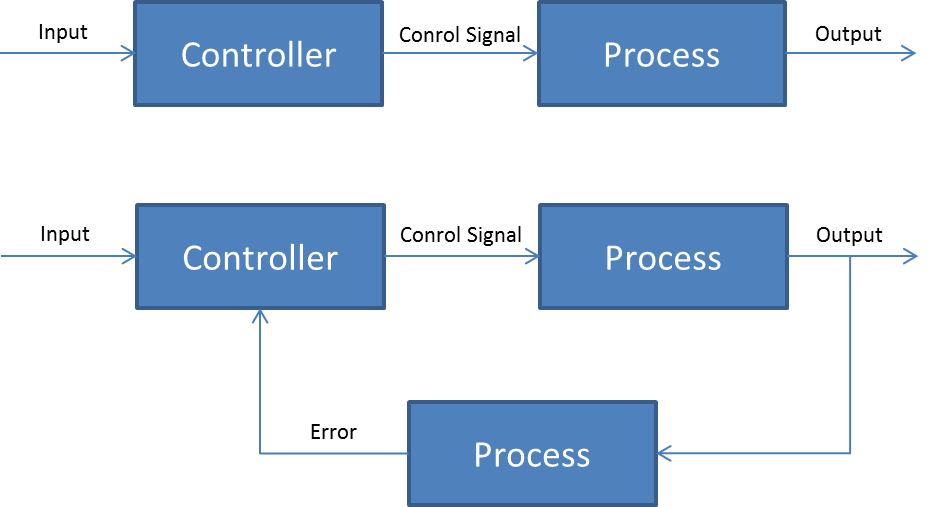
## 6.1 Basic Concept

Controllers are omnipresent in all kinds of dynamic systems, such as engineered processes or machines. The behavior of such systems is influenced by certain input parameters. A controller acts as a “corrector” by monitoring **process variables** (PV) and by comparing them with set points (SP)—which serve as references—to produce control actions in the form of a feedback mechanism. This feedback consists of the error signal between actual and target values and is also known as SP-PV error (Yu, 2006). The set of process sensors, controller functions, and the final control element is called a control loop. Generally, there are two types of control systems: open-loop and closed-loop systems. The previously described principle represents a closed loop and is also known as feedback control. An open loop, on the contrary, is created by leaving out the feedback signal, which is obviously a naïve approach because the control is done only based on the inputs and ignoring the output of the particular process (Goodwin et al., 2001). The following figure shows the general principles of open (top) and closed loops (bottom).

**Process variable**

Thiscaptures the value of a monitored part of a process.

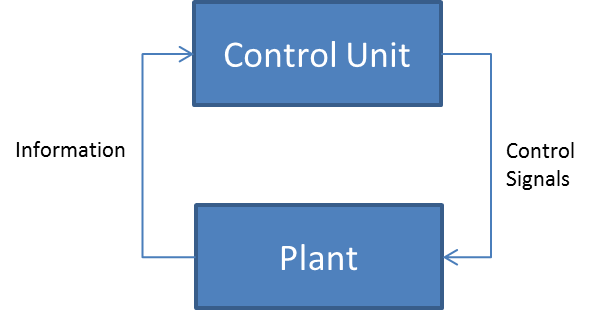
Principles of Open Loop (Top) and Closed Loop (Bottom)



Many processes in the industry do not contain just one single controller and control loop but, instead, consist of many individual ones that have to be monitored simultaneously. Supervisory control tackles that problem to allow an operator to get a proper overview of the overall process. In general, two different types can be differentiated—manual and automatic control (Sheridan, 1976). In manual control, an operator “tells the process” what should be done, i.e., directly manipulates the control elements. That kind of systems was the first approach to supervisory control and has mostly been replaced by automatic variants where the the process operates autonomously and the operator only monitors and makes adaptions from time to time.

In the last decades, different approaches to designing supervisory controllers for discrete event systems—such as automata, Petri nets, or timed Petri nets—have been investigated (Seatzu et al., 2013). The objective of a supervisory control can be described as the achievement of certain defined specifications by modifying the behavior of a system (plant G) using control units (supervisors S) (Cassandras & Lafortune, 2009). The principle is illustrated in the following figure.

Control System Consisting of Plant and Control Unit



Possible events in discrete event systems can be divided into controllable and uncontrollable (Balemi, 1992). As the name suggests, the former are events that can be managed by the supervisor, while the latter are events where no control authority is available, such as machine breakdowns and malfunctions, among others. Supervisory control is based on the supervisory control theory proposed by Ramadge and Wonham (1987), which formally describes the two main parts—the plant and the specifications—using formal languages.

## 6.2 Specifications

Regardless of the model used for the representation of a discrete event system, a specification of a supervisory control can be interpreted as follows. The language Lp representing all possible behaviors of a plant process G contains both admissible behaviors and behaviors that are either not acceptable, undesirable, physically inadmissible, or not allowed (Cassandras & Lafortune, 2009).

Unacceptable behaviors are, for example, violations of safety conditions while undesirable ones are behaviors that lead to some sort of deadlock in the process. A physically inadmissible behavior could be the placement of parts into a full buffer and an example for a behavior that is not allowed is the violation of a necessary order of events.

All those different types of behaviors of G that do not belong to the admissible category should be excluded by the supervisor S, i.e., the language Lp has to be restricted to the range between the maximal admissible and the minimal required behaviors La and Lr, respectively, where Lr⊂ La ⊆ Lp.

A plant process modeled as a discrete event system can be described by two languages Lp and Lm and formally defined as follows. (Awad, 2018).

A process G can be defined as the triple P=(Σ, Lp, Lm), where

Σ is an alphabet,

Lp is a language representing all possible tasks and is prefix-closed, and

Lm ⊆ Lp is the language representing all completed tasks, called marked language.

Lp and Lm are defined over the event set E=Ec∪Euc, where

Ec is the set of controllable events and

Euc is the set of uncontrollable events.

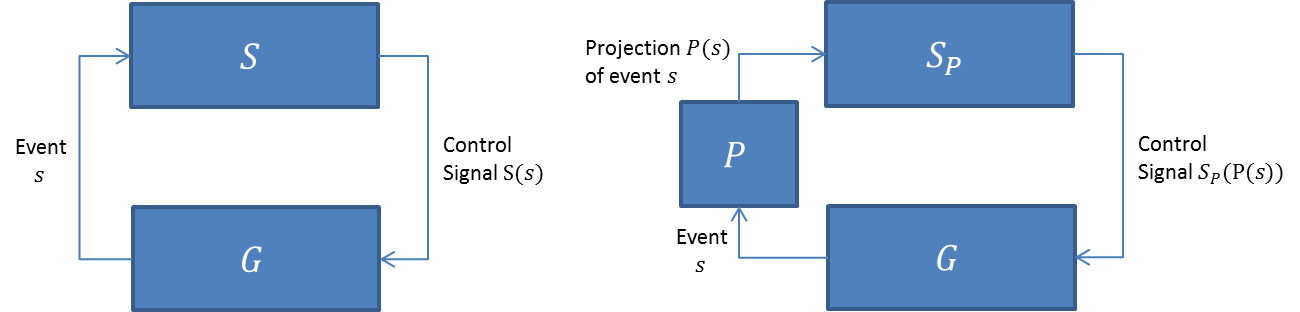
We call the controlled system S/G the system that is described by the two sublanguages Lp(S/G) and Lm(S/G) of Lp(G) and Lm(G), respectively, which only contain feasible strings in the presence of the supervisor S. These two sublanguages are generated and marked by S/G. Simply put, the language marked by S/G contains those marked words of G that still exist after introducing S. The aim of the specification of a supervisory control is now to restrict the behavior of G to a certain subset Lp(S/G)⊆La⊂Lp(G) or within a range of sublanguages Lr⊆ Lp(S/G)⊆ La⊂ Lp(G) by means of S.

This holds for cases where the supervisor can observe all events of the process, however, in cases where not everything can be “seen,” i.e., the event set can be divided into Eo and Euo (observable and unobservable events, respectively) where E=Eo∪ Euo. This is called partial observation (Cassandras & Lafortune, 2009) and it can be better understood by means of natural projection.

A natural projection proj[Σ’](L) of a language L on the alphabet Σ’ removes all symbols from the strings of L that are not included in Σ’. This means a projection from domain Σ to Σ’ (Feng & Wonham, 2008).

In the case of partial observation, the natural projection is obviously given by the projection from E to Eo. The following figure shows the feedback loop of supervisory control for a full (left) and partial observation (right).

Supervisory Control for Full (Left) and Partial Observation (Right)

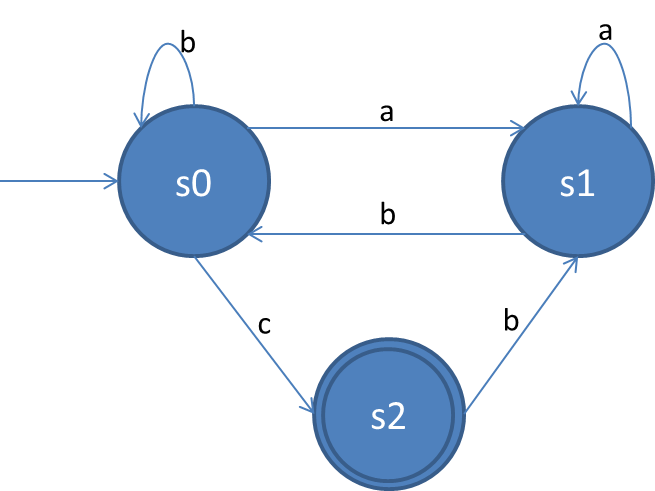


The easiest modeling of discrete event systems is done by assuming the process G to be a finite-state automaton and the associated languages to be regular (Cassandras & Lafortune, 2009). Thus, the specifications of the supervisory control can be modeled as automata. These have to be derived from the natural language specifications and can be formulated as basic instructions, such as “follow a first-in-first-out strategy” or “event a has to occur more often than event b.” Most of the natural specifications relevant in applications can be captured by simple automata—here denoted as Hspec—that generate the particular language requirements La. Depending on the particular events, one can use either product or parallel composition, and these are combined with the automaton G representing the process to obtain Ha, where L(Ha)=La (Cassandras & Lafortune, 2009).

Parallel composition is used when events that can occur in G but are not part of the transition diagram of Hspec are irrelevant for the particular specification. Instead, product composition is used when events that can occur in G are not part of the transition diagram of Hspec because they are not part of the La. For a summary of the different composition techniques, the reader is referred to the literature (Cassandras & Lafortune, 2009).

The following two common types of specifications serve as easy examples for the modeling by means of automata. We use the following model of the discrete event system G to show how the techniques work.

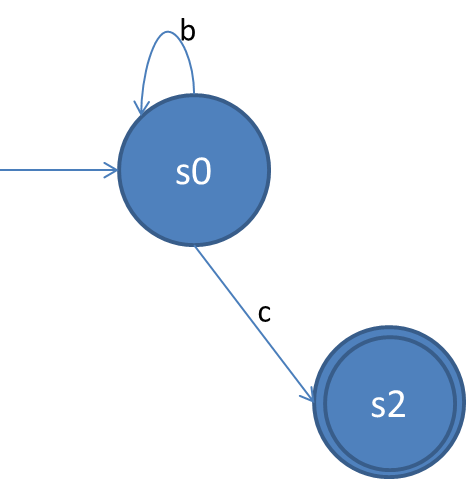
Automaton for Exemplary Process G. State s2 Marked



Specifications that define states of G to be illegal are modeled by considering the accessible part of G after removing the illegal states and the associated transitions. This can incorporate blocking behaviors, which means that it is not possible to reach a marked state from every reachable state. Should that be permitted, the part of G that is both accessible and **co-accessible**—known as the TRIM operation—is considered. The following example shows the incorporation of a specification for G defining state s1 to be illegal.

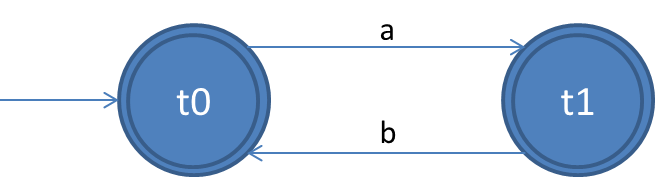
**Co-accessible** A state of an automaton is called co-accessible if there is a sequence of transitions that leads to a marked state.

Resulting Automaton Specifying s1 to Be Illegal



A specification defining an event alternation is easily modeled by means of a two-state finite automaton, as illustrated in the figure below with the event set {a,b}, where a is the first event to occur.

Automaton Modeling State Alternation



Both states are marked to generate non-blocking behavior. Ha is obtained by applying parallel composition, which does not affect other states of G except for a and b and does not allow a second occurrence of a until b. Thus, L(Ha)= Hspec||G.

To repeat the definition of parallel composition, as mentioned before,

with

where the composition of two states is represented by a point and which leads to the following transitions

δ(s0.t0,a)=s1.t1,

δ(s0.t0,c)=s2.t0,

δ(s0.t1,c)=s2.t1,

δ(s0.t1,b)=s0.t0,

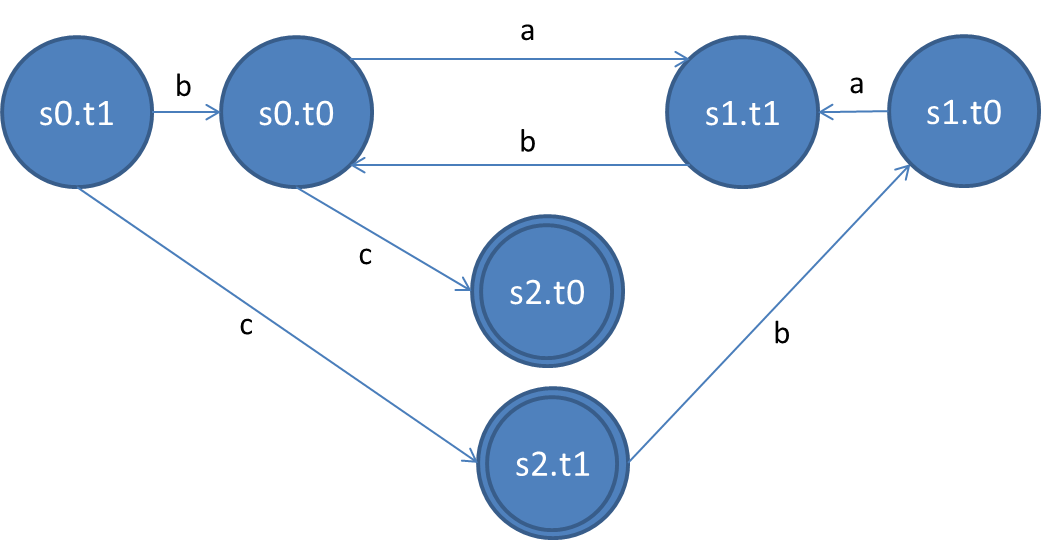
δ(s1.t0,a)=s1.t1,

δ(s2.t1,b)=s1.t0,

δ(s1.t1,b)=s0.t0.

In the transitions shown above, the marked states are underlined. This can be recognised by the automaton shown below.

Automaton Representing State Alternation



An important thing to consider is that the combination of a marked and an unmarked state is unmarked. It can be seen that the possibility for G was successfully removed to consecutively execute events a and b twice.

The incorporation of other specifications, such as state splitting or illegal substrings works in a similar way. The interested reader is referred to the literature offering more examples of this type.

## 6.3 Synthesis

The aim of supervisory control synthesis is the derivation of a supervisor that incorporates all required specifications without further restrictions of the particular plant. Formally, a supervisor S can be defined as a function from the language generated by the plant G to the power set of the event set E

Where does this definition come from? In the control **paradigm** of supervisory control, the supervisor can dynamically enable and disable all controllable events of E, i.e., for each controllable event there are two possible states. We already saw that the parallel or product composition of an automaton G representing a plant and an automaton Hspec representing a specification yields the desired automaton Ha that incorporates the specified behavior. Such a specification is called uncontrollable if a word of Lp(G) that is generated according to specification Hspec yields a prohibited behavior when followed by an uncontrollable event. Conversely, a specification is controllable if the word that leaves the permitted behavior does not exist (Morgenstern & Schneider, 2007). In the case of uncontrollable specifications, the resulting automaton Ha contains bad states with undesired behavior. A more formal definition of a supervisory control problem can be found below.

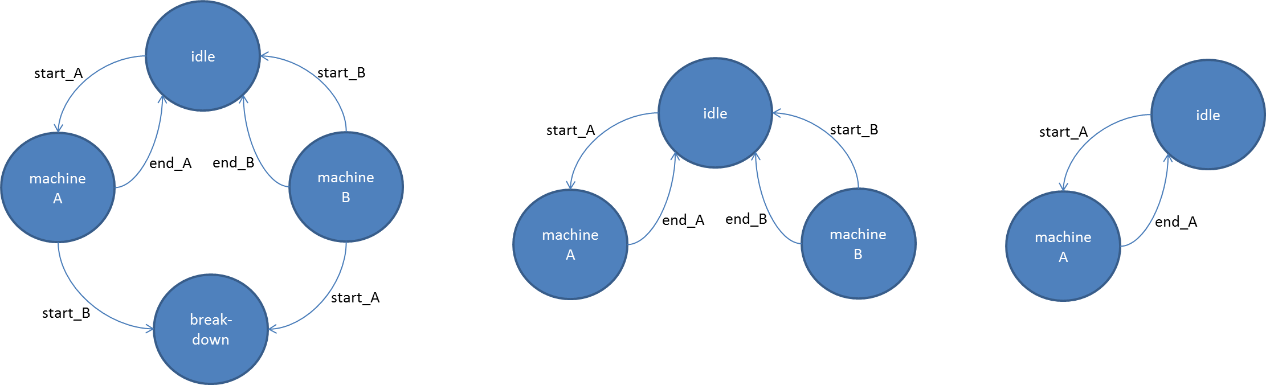
**Paradigm**

A certain set of concepts forming thought patterns.

Given a plant G and a specification language Lam⊆Lp(G) that defines the desired behavior under supervision, find a non-blocking supervisor S so that

Thus, we obtain the least restrictive solution for a supervisor by combining the plant and the specification automaton. Nevertheless, the non-blocking behavior of a controlled plant is not enough to guarantee that a marked state will be reached because it does not limit the length of the sequence of events leading to it, i.e., a state is co-reachable even if the sequence is infinite. This becomes clear considering the following example.

Uncontrolled Plant (Right), Desired Supervised Behavior (Center), Controller with Blocking Behavior (Right)



On the left side of the figure you can see a plant without controller, which consists of four states, where “idle,” machine A, machine B and “breakdown” represent the initial state, activated machines A and B, and a breakdown state, respectively. Starting from the initial state, the events start\_A and start\_B start the respective machine. If the particular task is finished, the plant switches back to the initial state via the uncontrollable events end\_A and end\_B. If a start signal for a machine is set while the other one is executing its task, the plant enters a breakdown state, which cannot be left anymore. The supervised behavior can be seen in the figure at the center, where the undesired state and the associated transitions are removed. We already showed how the removal of undesired states works. Another problem, in fact, is the implementation of a deterministic controller that selects only one of the signals start\_A and start\_B at a time. On the right side of the figure is a blocking controller, which always selects signal A, which means that machine B will never be selected by the controller. From the controller’s perspective state machine B is still co-reachable but the sequence leading to it may be infinite. Thus, the non-blocking property may not be violated. This leads us to the definition of forcibly co-reachable states, which is a more rigid criterion (Morgenstern & Schneider, 2007). A state is said to be forcibly co-reachable if a marked state is reached after a sequence of events of finite length, i.e., if there is a threshold for the length of the sequence. Since the formal definition of this property is rather complex, we want refer the reader to the literature (Morgenstern & Schneider, 2007). An automaton is called forcibly non-blocking if all of its reachable states are forcibly co-reachable. After introducing this concept, we can now have a look at the formal definition of the controller synthesis problem.

Given a plant G and a specification language Lam⊆Lp(G) that defines the desired behavior under supervision, find a non-blocking supervisor S so that

- the composition of S and G is forcibly non-blocking.

By means of that definition we can easily derive a deterministic controller that ensures that a marked state will be reached. Since we know that all events that occur from a forcibly co-reachable state will reach a marked state at some point, our controller can select any controllable event in each time step. For more details on the different approaches to the implementation and further insight into the necessary steps of the algorithm, the interested reader is referred to the literature (Morgenstern & Schneider, 2007).

## 6.4 Performance Analysis

During the synthesis of supervisory controllers, it is necessary to analyze and evaluate their performance. Different performance criteria can be of interest, such as the average throughput, the work in progress, or wait time (Košecká, 1992). A more formal qualitative comparison between multiple discrete-event system controllers is the concept of permissiveness. Maximal permissiveness means that a supervisor only disturbs the plant when it is absolutely necessary. It is proven that a unique maximally-permissive supervisor exists for a plant G if there is a non-blocking and safe supervisor for G (Ehlers et al., 2013). However, maximal permissiveness does not necessarily mean best performance (Wang & Ray, 2004; Cassandras & Lafortune, 2009). Wang and Ray (2004) propose a method for the quantitative performance evaluation of discrete event systems based on a signed real measure of regular languages.

Several studies are directed towards the application of different types of Petri nets for the modeling, control, and performance evaluation of discrete event systems (Čapkovič, 2017). The necessary properties, such as non-blocking behavior, can thus be easily be evaluated (Giua & Silva, 2017).

**Trade-off**

This is an opposed dependency of two parts. If one part improves, the other one worsens, and vice versa.

Kaymakci and Kurtulan (2009) propose a new performance measure depending on numeric values that are derived from strings corresponding to blocking and success. An optimization approach is used to find a **trade-off** between blocking behavior and the success of a supervisor.

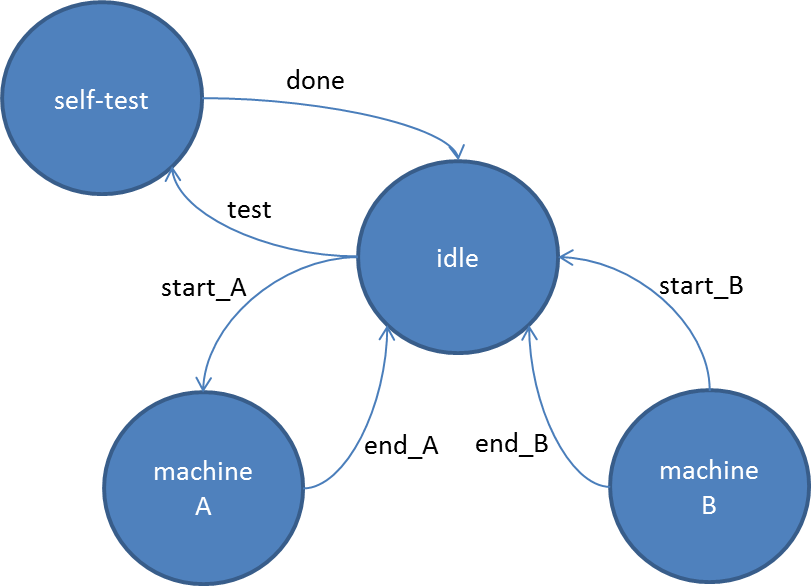
Given the high degree of complexity and diversity of this topic, this section merely serves as an overview of existing approaches and the interested reader is referred to the literature for further details.

## 6.5 Implementation

In the case that a plant generates all events of its event set—controllable and uncontrollable ones—on its own or by means of an agent, the implementation of a supervisor is simple because it only has to enable and disable the controllable events (Dietrich et al., 2002). This scenario does not represent the reality because most events are controllable ones that have to be initiated, which cannot be done on its own but, for example, by giving a start command using a switch. The designed supervisor should be able to initiate such events in addition to the enabling and disabling of all controllable ones. We denote the set of events that should be initiated as Ξ and, obviously, it holds that Ξ⊆Ec.

Furthermore, we can define a Ξ–implementation of a plant G, which only restricts the occurrence of events in Ξ by enabling a maximum of one event contained in it if it exists in G (Dietrich et al., 2002). To visualize that rather abstract definition let us have a look at an example, which is an extension of the two-machine scenario introduced earlier.

Implementation-Dependent Controlled Plant G



The figure shows the controlled plant G. Again, we have two machines that can be started with the controllable events start\_A and start\_B and that finish with the uncontrollable events end\_A and end\_b, respectively. While being in states machine A and machine B, the events start\_B and start\_A would lead to a breakdown state which cannot be left anymore. Additionally, we included a “self-test” state, which can be accessed via the controllable event “test” when in state “idle” and is left with the uncontrollable event “done.” When G is in state “self-test” no machine can be started. The supervisor at the beginning of this unit was able to select start\_A and start\_B and, thus, forbid the plant’s transition to the breakdown state.

Now, we define the set of events that should be initiated by our supervisor as Ξ={start\_A, start\_B}, since these events cannot be generated by the plant itself or by means of an agent. In our case, the controllable event test is generated by a human serving as an agent and our supervisor only disables it when being in states machine A or machine B but never selects it as the one which should occur next. A Ξ–implementation would be a supervisor that enables the events start\_A and “test” in state “idle” and no events in states “self-check,” machine A, and machine B. If event “test” would be disabled when being in state “idle,” the supervisor would be no Ξ–implementation because test∉ Ξ. The same holds for disabling events start\_A or start\_B when being in state “idle.” Events from Ξ have to be allowed to occur but would be prohibited by the supervisor in this case.

Termination and confluence are two properties that have to be met for an implementation to be non-blocking. A terminating system stabilizes after a sufficiently long time, i.e., there is a sequence of events of finite length after some input is given (Dietrich et al., 2002). If G is Ξ-terminating there is only a finite sequence of events of Ξ allowed to occur. It is then clear why systems that do not correspond to that criterion create problems during implementation. If the number of events that should occur next and that are chosen by the supervisor is infinitely large, an implementation would be stuck inside a loop.

The concept of confluence is a bit more difficult to explain. A confluent system ensures that states with the same future are reached by all implementations by the occurrence of events from Ξ only, independently from a chosen event. For the formal definition of this criterion, we refer the reader to the literature (Dietrich et al., 2002). The two properties can now be summarized as follows: If G is both Ξ-terminating and Ξ-confluent, then states with the same future will be reached by all Ξ-implementations starting from a reachable state for all events of Ξ.

There are several software tools for the implementation of supervisory control of discrete event systems. Pinheiro et al. (2015) proposed “Nadzoru,” which provides a graphical user interface for a convenient creation, simulation, and analysis of systems. Ricker et al. (2006) developed a tool called “DESUMA,” which serves for the study of discrete event systems based on finite-state automata. Other examples are “Grail” (Reiser et al., 2006), “IDES” (Rudie, 2006), or “STS” (Gu et al., 2018).

|  |
| --- |
| Summary |
| In the last few decades, different approaches to the design of supervisory controllers for discrete event systems have been investigated, such as automata, Petri nets, or timed Petri nets. The objective of a supervisory control can be described as the achievement of certain defined specifications through the modification of the behavior of a plant by means of supervisors; it is based on the supervisory control theory. A supervisor’s task is to enable and disable certain events according to specifications that define the admissible behavior of a system. Specifications can be represented as finite automata that are merged with the finite automaton model of a plant using product or parallel composition to yield the desired admissible behavior.  Blocking should be prevented because it is a state which cannot be left by enabling or disabling an event. However, the non-blocking behavior of a controlled plant is not enough to guarantee that a marked state will be reached because it does not limit the length of the sequence of events leading to it. Thus, an infinitely long sequence could be considered non-blocking. Because of that, the property of forcible co-reachability is also considered in the formulation of the controller synthesis problem. A state is forcibly co-reachable if a marked state is reached after a sequence of events of finite length. An automaton is called forcibly non-blocking if all of its reachable states are forcibly co-reachable.  Maximal permissiveness is a type of qualitative comparison between multiple discrete-event system controllers and it means that a supervisor only disturbs the plant when it is absolutely necessary.  In the case that a plant generates all events of its event set autonomously or by means of an agent and since it only has to enable and disable the controllable events, the implementation of a supervisor is rather simple. In general, most events are controllable and have to be initiated, which should be done by a supervisor in addition to enabling and disabling the set of controllable events. Termination and confluence are two properties that have to be satisfied for an implementation to be non-blocking. |

# Unit 7 – Applications

#### Study Goals

On completion of this unit, you will have learned …

… how discrete event systems can be used to supervise production systems.

… how monitoring and fault diagnosis can be established.

… the basics of decentralized and distributed supervision.

… what kinds of applications exist for the model-based optimization of production systems.

… the basics of adaptive supervisory control.

# 7. Applications

## Introduction

Discrete event systems are not only used in theory but can be found in all kinds of real-world applications. This unit will provide an overview of different scenarios where such systems can be found. The supervision of production systems can be separated into monitoring, supervision, and control tasks and different approaches exist regarding how those parts can be connected with each other. Faults play an important role in all kinds of technical systems and the monitoring and detection of those is essential to guarantee proper functionality. The decomposition of a monolithic supervisor following a decentralized or distributed paradigm is an approach for complexity reduction in the context of supervisory control. This unit will also introduce the basics of the two paradigms and explain how they can be compared to each other. Additionally, the unit will provide an overview of the existing studies on different discrete event systems that can be used for the optimization of production systems and present the concept of adaptive supervisory control.

## 7.1 Production System Supervision

One example for a real-world application of discrete event systems is the supervision of production systems with a special focus on the failure aspect, such as unforeseen and uncontrollable malfunctions (Combacau et al., 2000). The main difficulty in the implementation of such a system lies in the necessity to be able to recognize situations showing abnormal behavior and to find its origin.

In general, two different types of failure handling can be differentiated—failure avoidance and failure processing (Combacau et al., 2000). The aim of the former approach is try to foresee failures and keep the probability of their occurrence as low as possible, while the latter aims at foreseeing and integrating the processing of some failures into the controller, as well as at incorporating monitoring and supervision systems.

In the case of a normal operation, the supervision enables and disables the controllable events. Its role during the occurrence of failures, on the contrary, is the determination and execution of the necessary steps to bring the system back to its normal operation state, which, for example, can be done by implementing emergency or recovery actions.

The two other important parts needed for the implementation of production systems supervision are control and monitoring. The former refers to the execution of control signals for the actuators of a system, i.e., all functions that have direct influence on the underlying process, while the latter is the collection of process and controller data without direct action on the model or process. Let us have a look at the definition of some essential terms before we continue the discussion on the supervision of production systems (Combacau et al., 2000).

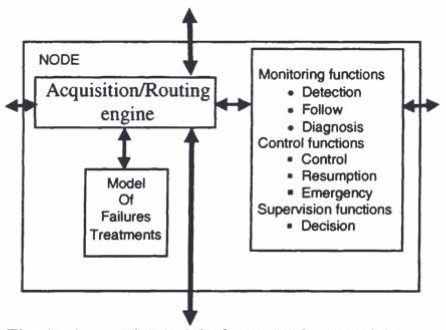
Terms in the Context of Supervision of Production Systems

|  |  |
| --- | --- |
| Fault | Action that violates specifications (intended or unintended) |
| Defect | Deviation between actual and nominal value of a parameter |
| Error | Violation of the specification of the system (part of model) |
| Latent error | Error that did not have any impact on the system’s operation yet. Becomes effective error after usage |
| Malfunction | Deviation from expected behavior after executing an operation |
| Failure | Event that defines a situation where an operation is not executed due to a deviation of state from the defined specification |
| Breakdown state | State resulting from failure, which does not allow specified service |
| Symptom | Information representing abnormal behavior |
| Exception | Defined execution of an action after recognition of a symptom |
| Recovery point | State that can be reached from breakdown state from where the system has to be brought into normal operation state |
| Recovery sequence | Sequence containing necessary actions to bring the system from a breakdown to a recovery state |

Using a combination of monitoring and supervision, production planning can be implemented in both normal and abnormal operation modes. One can differentiate such systems between online, a term which refers to the reconfiguration during the exploitation phase after failure detection, and off-line, which refers to the investigation of strategies to eliminate the impact of failures. The latter incorporates the development of an accurate and reliable discrete event system model capturing normal and abnormal behaviors, where failures are modeled as uncontrollable events. In this context, the goal of monitoring and supervision is the prediction of those events based on observable events. A controller should be synthesized so that the detection of a failure leads to a degraded functioning mode, which is kept until normal operation is reachable (Combacau et al., 2000).

Hierarchical supervision and monitoring represents an approach where the monitoring and supervision system incorporates, among others, the underlying manufactured products and the specified production policy in addition to the detected failures to obtain a reaction. The three parts—monitoring, control, and supervision—are composed in a modular way, which allows for a more flexible processing of failures. While monitoring takes care of detection and diagnosis, supervision takes decisions and the controller executes control, resumption, or emergency actions (Zamaï et al., 1998). The following figure shows the concept of the hierarchical approach.

Model of Hierarchical Supervision and Monitoring



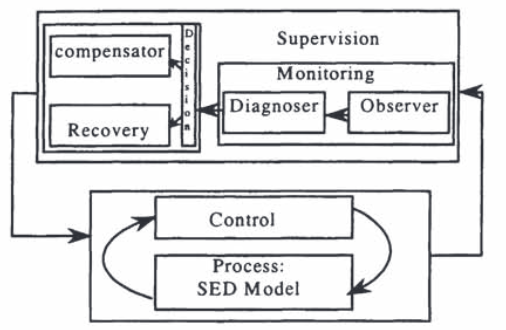
The acquisition/routing engine takes care of the communication between the different functions of the system by distributing the incoming data. While Petri nets are often used for the modeling of control functions, **extended entity-relationship-models** are applied to collect temporal information (Thalheim, 2009).

**Extended entity-relationship-models**

By means of these models, the data requirements of information systems can be described graphically.

Another approach to the stabilization of discrete event systems uses an architecture with two levels. The lower level contains the production system and the associated controllers providing nominal functioning, while the higher one represents the supervisor that takes over the control in case of a failure. The supervision itself can be divided into monitoring and decision-recovery-compensatory modules. A visual representation of this architecture is provided in the following figure.

Architecture for Supervisory Control of Discrete Event Systems



A recovery action is executed in the case of a failure and it decides the next step, namely, either a return to nominal functioning or the entrance into degraded functioning mode, where the system stays until it can re-enter the nominal state (Combacau et al., 2000). Since recovery actions incorporate the past history of the low-level system in addition to the type of detected failure, the monitoring module needs diagnostic skills in order to be able to identify the type of failure that occurred.

The supervisory control ad data acquisition (SCADA) is a system for the monitoring and control of production, or of industrial processes in general (Boyer, 2009). By means of a combination of software and hardware components, such systems allow the collection and analysis of data, monitoring, and control of systems. SCADA is used in all types of industries, such as food and beverage processing (Holmes et al., 2013), pharmaceuticals (Rajeswari et al., 2013), energy pipelines (Irannejad & Iraninejad, 2014), energy management (Figueiredo & da Costa, 2012), or wastewater treatment (Humoreanu & Nascu, 2012). For further information on this particular system, the reader is referred to the literature (Boyer, 2009).

## 7.2 Monitoring and Diagnosis of Faults

As mentioned earlier, monitoring plays an essential role in the supervision of all kinds of industrial systems. Its aim is to collect and process all possible observations generated by the underlying system, i.e., all observable events. Closely connected to monitoring is the diagnosis of faults, which is responsible for fault detection and fault isolation. While the former indicates the detection of a fault, the latter refers to the search for its origin. In the context of monitoring and fault diagnosis, the model of discrete event systems can be formally defined as follows using the finite-state automata framework (Chanthery & Pencolé, 2009).

Given a system G represented by the tuple (K,Σ,δ,s0), where

K is the finite set of states,

Σ is a finite set of events,

δ is a finite set of transitions, and

s0 is the initial state,

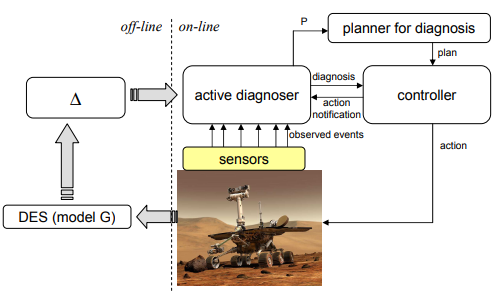
the set Σf ⊆ Σ represents the set of faults of G. Monitoring can be defined as the function Σ→( Σo∪{ε}), i.e., it associates an observable or empty event to each event e∈Σ.

In general, the fault diagnosis problem for discrete event systems can be described as the provision of a set of all possible faults for a recorded sequence of observations σ (Chanthery & Pencolé, 2009). If there is at least one sequence of events ω in G that contains a fault F and for which the sequence of observable events Obs(ω) equals the sequence of recorded observations σ for F, Obs(ω) is called trace of F. The set of traces Trc(F) of F consists of all sequences where Obs(ω), i.e., F is a solution for the diagnosis problem if σ∈ Trc(F). Since Trc(F) is a regular language, it can be represented as a finite-state automaton by M(F)=(K,Σo,δ,s0,tag), where tag: K→{F-possible, F-impossible} (Chanthery & Pencolé, 2009).

Thus, the automaton assigns one of the tags F-possible and F-impossible to each of its states. M(F) and its counterpart M(¬F)—representing the particular automaton that captures the set of traces for no occurrence of F—can be used for the monitoring of a discrete event system. The decision regarding whether a sequence of observations is a trace of fault F or not can be made at any time. The parallel composition of the set of automata for ∀F∈Σf yields the classic diagnoser (Pencolé et al., 2006).

An extension of the classic interpretation of a diagnoser called the active diagnoser is proposed by Chanthery and Pencolé (2009). Besides the diagnosis for observable situations, this approach additionally evaluates the benefit of an active diagnostic session and incorporates a planner for the possible refinement of the diagnosis. The following figure shows the concept of the proposed active diagnosis.

Architecture of Active Diagnosis



Here, Δ represents the model of the active diagnoser. This setup allows an on-line adaption of the diagnostic behavior. To gain a deeper understanding of this interesting method, the reader is referred to the literature (Chanthery & Pencolé, 2009).

Besides finite automata, several approaches using different variants of Petri nets exist for the monitoring and diagnosis of discrete event systems, such as time Petri nets (Ghazel, 2011), partially observed (Ru & Hadjicostis, 2009), bounded (Ran et al., 2017), or labeled Petri nets (Cabasino et al., 2011; Cabasino et al., 2013). Furthermore, **statecharts** and **hierarchical state machines** are often used for modeling (Su & Wonham, 2006; Idghamishi & Zad, 2004). For a detailed overview and a comparison of different fault diagnosis methods for discrete event systems, the reader is referred to the literature (Zaytoon & Lafortune, 2013).

**Statechart**

Thisrefers to the graphic representation of finite automata.

**Hierarchical state machines** These are an extension of classical finite automata using hierarchically nested states.

## 7.3 Distributed and Decentralized Supervision

When supervisory control is applied to large and complex systems with many subcomponents and diverse specifications, it is generally not advisable to use a **monolithic** supervisor, but, instead, to decompose it in order to improve the computational feasibility and transparency of the particular control actions. This is known as a decentralized approach for supervision and consists of a division of the supervisory task into multiple subtasks that are supervised in a decentralized manner (Wonham & Cai, 2019). To obtain a supervision of the complete system the single decentralized supervisors work concurrently. An obvious possible disadvantage of this approach is the risk of conflicts and, as a result, a blocking behavior. We can illustrate the decentralized supervision as shown in the following figure.

**Monolithic**

In this kind of architecture, different functional aspects are not separated into multiple components, but rather intermixed with each other.

Architecture of Decentralized Supervision

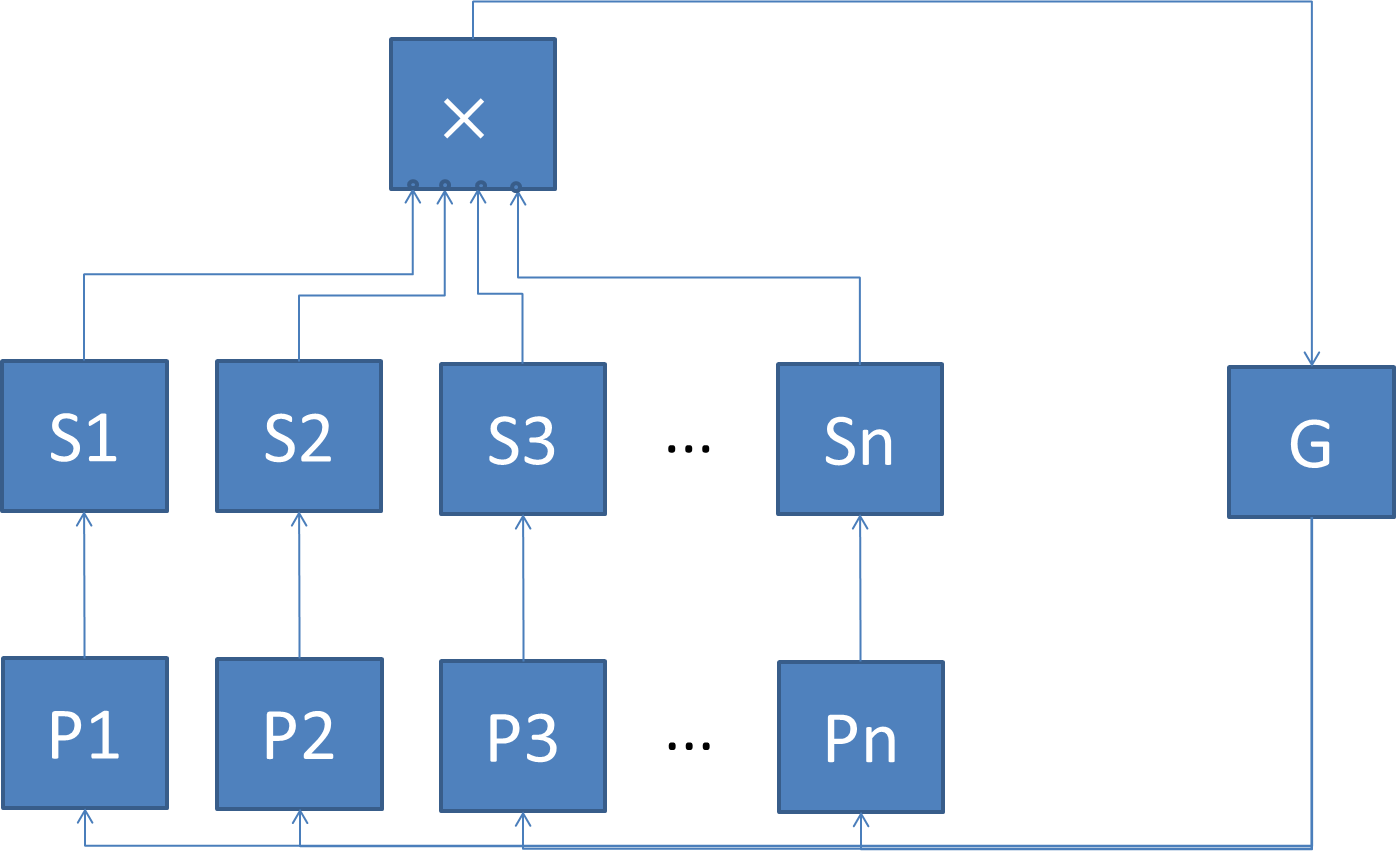


In this case, the supervisory task of the plant G is decomposed into two modular supervisors S1 and S2, which are constructed by means of decentralized modular synthesis (Wonham & Cai, 2019). Such systems are easier to maintain and to modify due to the reduced complexity of the single supervisors, i.e., if a subtask of G is changed, only the particular supervisor has to be adapted, while the remaining ones remain unchanged. Due to the previously mentioned problem of possible conflicts and blocking associated with it, one of the main tasks of decentralized supervision is to guarantee a non-blocking behavior of the overall synthesis.

The conjunction of two supervisors S1 and S2 for G, represented by S1∧ S2, can be defined as the reachable (accessible) part of the product composition

The reachable part of an automaton (also called reachable sub-automaton) is the resulting automaton after removing states that can never be reached and the associated transitions. Following the definition of product composition, an event is only enabled if and only if it occurs both in S1 and S2. We can schematically represent the decentralized supervision principle as illustrated below.

Decentralized Supervision: Architectural Design



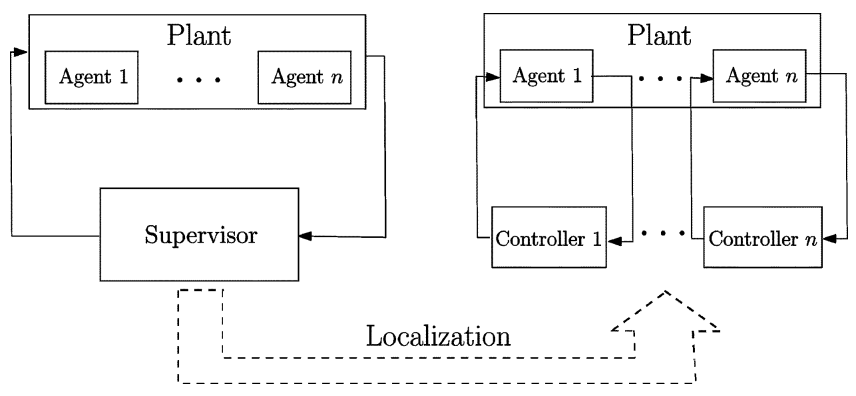
**Natural projection**

This is the projection from one domain to another. It is also known as homomorphism.

Each supervisor Si with 1≤i≤n only sees a part of the behavior of the discrete event system G, i.e., it only processes a **natural projection** Pi with 1≤i≤n of the behavior.

Although the two are related to each other, distributed supervision must be clearly distinguished from the decentralized approach. While the latter follows a supervisor-subordinate paradigm by allocating the global control action among modular supervisors that enforce local specifications, the former uses a distribution among individual agents by means of the decomposition of a synthesized monolithic supervisor into local controllers (Cai & Wonham, 2010a). An algorithm for that procedure—called supervisor localization—was proposed by Cai and Wonham (2010a) and it was proven to have the same global behavior as the underlying monolithic supervisor. The figure below illustrates the concept of supervisor localization.

Supervisor Localization



On the left side of the figure a plant with n active components (agents) is shown under supervision. The left side illustrates the concept of allocating the global control among n local controllers that individually control one of the n agents. Here, the difference between distributed and decentralized supervision becomes clearer: While the decentralization yields a smart supervision of components that follow directions blindly, distribution yields smart agents. The difference can be enhanced in a more illustrative manner by highlighting the fact that, while decentralized supervisors are located externally, distributed supervisors are built into the active components of a plant.

For a detailed introduction of these research topics, the interested reader is referred to the literature (Cai & Wonham, 2010a; Cai & Wonham, 2010b; Wonham & Cai, 2019).

## 7.4 Model-Based Optimization of Production Systems

Model-based optimization is a concept used to control the quality of products in production or manufacturing systems (Thombansen et al., 2018). It mainly consists of the monitoring of the particular plant of interest, which can be defined as the association of an observable or empty event to each event e∈Σ. Formally, this can be represented as the function Σ→( Σo∪{ε}). Thus, monitoring keeps track of the system’s behavior without interacting with it and communicates with other components—such as diagnosers—to detect and analyze faults.

Petri nets are often used for the modeling of production systems where the production of desired materials can be affected by a reachability problem of the net (Gyapay et al., 2002). Piera et al. (2004) proposed an optimization technique of logistic and manufacturing systems by means of simulation of colored Petri nets. Instead, Zhang et al. (2017) used Petri nets for the optimization of scheduling of crude oil operations. Finally, the use of Petri nets in order to avoid deadlocks in automated manufacturing systems was also proposed by Xing et al. (2008).

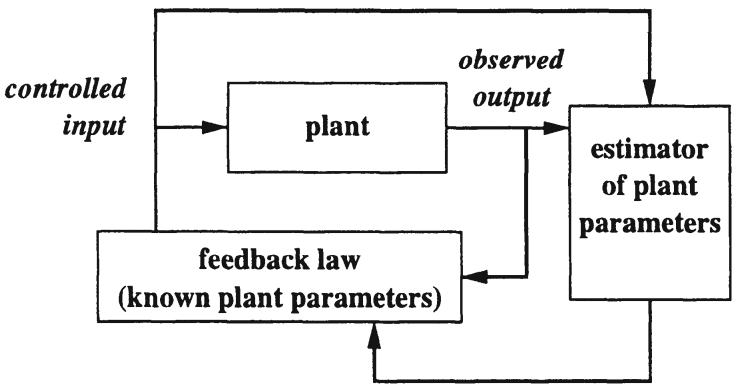
Other studies make use of the automata framework for optimization of production systems. Panek et al. (2006) proposed the optimization of timed automata to efficiently synthesize production schedules. A very interesting approach for throughput optimization of partially-controllable manufacturing systems and supervisor synthesis was recently published by van Putten et al. (2020). In this study, they use of extended finite automata, which are, as the name suggests, extensions of the classic automaton model (Alagar & Periyasamy, 2011). While transitions of classic automata are associated with sets of Boolean values, their extension uses trigger conditions to transit from one state to another. Reveliotis and Nazeem (2014) proposed an approach which used finite state automata in automated manufacturing systems to avoid deadlocks.

For more detailed investigations of model-based optimization techniques for production systems, the interested reader is referred to the literature.

## 7.5 Adaptive Supervisory Control

Adaptive supervisory control represents a useful paradigm for complexity reduction of supervisory controller synthesis for discrete event systems with partial observations by assuming a decomposition of the control design into two parts (Boel, 2002). Formally, a partial observation means that there is a natural projection from the set of events E to the set of observable events Eo of a plant G. There are different modes of operation in a plant that can be differentiated; the task of the first component of the control loop is the estimation of whether the set of all those modes is compatible with the observations collected in the past. The feedback controllers corresponding to the estimated modes of operation are activated by the second component (Boel, 2002). The general structure of an adaptive control loop can be seen in the figure below.

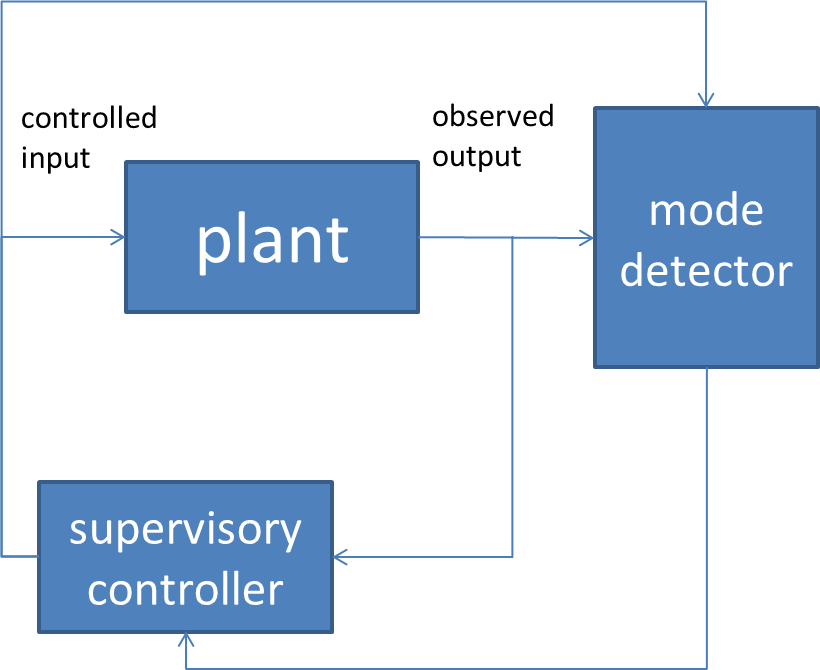
Adaptive Control Loop



Similar to the goal of general supervisory control, the aim here is to keep the plant at a safe state independently from the occurring sequence of uncontrollable events. The motivation behind this approach is the assumption that the parameters of many real-world plants change slowly, reducing the size of the plant’s state space and consequently its complexity (Boel, 2002).

To fully understand the concept of modes of operation it might be useful to consider the following example of a plant in which failures occasionally occur and are repaired after a certain amount of time. While failures are generally uncontrollable and unobservable events, the repairing process can be observed and controlled (Boel, 2002). Following the classic paradigm of supervisory control, different supervisory controllers may have to be synthesized for each status of the plant. The adaptive supervisory control paradigm now assumes that the particular current state remains as it is forever, which dramatically simplifies the specifications. Different modes of operation MOi can be derived by representing the status of each component of the plant, which are estimated by the first component of the adaptive supervisory control loop. The second component of the loop applies the particular controls by enabling and disabling the events associated to the particular mode of operation. In the following figure, the general adaptive control loop is changed to represent the architecture of adaptive supervisory control.

Control Loop of Adaptive Supervisory Control



The mode detector is modeled as a timed discrete event system, e.g., a timed Petri net, that maps the timed sequence of past observations σ={(ek,τk), k≤n}. In this system, τk is the time value for the k-th event ek to the set of modes of operations Rt⊂MO compatible with σ at time t, where MO represents the set of all possible modes of operation (Boel, 2002). **Reachability graphs** are used for the estimation of Rt. The main issues for adaptive supervisory control are the appropriate selection of the underlying plant’s state space and the partitions of the set of modes of operation.

**Reachability graph**

This shows all markings of a Petri net that are reachable by triggering enabled transitions.

This adaptive paradigm has successfully been used by Xu et al. (2009) for the control of fuel cell/battery-powered city buses. Several studies explore the incorporation of adaptive supervisory control into autonomous systems. Davidrajuh (2012) proposed a design of fault-tolerant autonomous systems using adaptive supervisory control. Related to that, Shiomi et al. (2009) employed adaptive supervisory control for a communication robot that approaches visitors. For further information, the interested reader is referred to the literature.

|  |
| --- |
| Summary |
| Discrete event systems are not just a theoretical concept but can be found in many real-world applications. One example is the supervision of production systems with a particular focus on failures, such as unforeseen and uncontrollable malfunctions. The main difficulty in the implementation of such a system lies in the necessity to be able to recognize situations showing abnormal behavior and to find the origin of said behavior.  More generally, two different types of failure handling can be differentiated—failure avoidance and failure processing. The former tries to foresee failures and to keep the probability of occurrence as low as possible, while the latter attempts to foresee and integrate the processing of some failures into the controller and to incorporate monitoring and supervision systems. A composition of monitoring and supervision production planning can be implemented in both normal and abnormal operation modes, while on-line and off-line systems can be differentiated.  Fault handling in systems requires the combination of monitoring and diagnosis. While the former refers to the detection of a fault, the latter refers to the search for its origin. The general purpose of a fault diagnoser is the generation of a set of possible faults that matches to the past events observed.  Decentralized and distributed supervision are two approaches to reduce complexity by means of composition of a monolithic supervisor. The latter follows a supervisor-subordinate paradigm by allocating the global control action among modular supervisors that enforce local specifications, while the former uses a distribution among individual agents by means of the decomposition of a synthesized monolithic supervisor into local controllers. Adaptive supervisory control splits the control loop into two parts: The first acts as an operation mode detector and the second as the supervisory control that activates those controllers that match the estimated mode. |

# Appendix 1 – Literature

Åkerman, M. (2018). *Implementing shop floor IT for Industry 4.0* [Doctoral dissertation, Chalmers University of Technology]. Chalmers Research.

Alagar, V. S., & Periyasamy, K. (2011). *Specification of software systems* (pp. 105—128). Springer.

Alyoukhin, A. G., & Silaev, A. A. (2019, March). Finite automata as control model for manufacturing discrete type system. *Proceedings of the 2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)* (pp. 1—6). IEEE.

Alur, R., & Dill, D. L. (1994). A theory of timed automata*. Theoretical Computer Science*, *126*(2), 183—235.

Amonkar, S. (2019, February 13*). Predictive maintenance and monitoring using AI and machine learning*. BrightTALK.

Autebert, J. M., Berstel, J., & Boasson, L. (1997). Context-free languages and pushdown automata. In G. Rozenberg & A. Salomaa (Eds.), *Handbook of formal languages* (pp. 111—174). Springer.

Awad, H. (2018). Supervisory control systems: Theory and industrial applications. In R. Campos-Rodriguez & M. Alcaraz-Mejia (Eds.), *Petri nets in science and engineering.* IntechOpen.

Bachelet, B. (1998). *Creation of libraries of reusable model components for the visual simulation environment* [Doctoral dissertation, Université Blaise Pascal]. ResearchGate.

Balemi, S. (1992). *Control of discrete event systems: Theory and application* [Doctoral dissertation, Swiss Federal Institute of Technology Zürich]. ETH Zürich.

Banks, J. (2005). *Discrete event system simulation*. Pearson Education.

Ben-Assa, B. (2019, September 11). The tremendous power of AI in manufacturing optimization—5 examples that clear the mist. *Plataine*.

Boel, R. K. (2002). *Synthesis and control of discrete-event systems* (pp. 115—123). Springer.

Bollobás, B. (2012). *Graph theory: An introductory course*. Springer.

Boyer, S. A. (2009). *SCADA: Supervisory control and data acquisition.* International Society of Automation.

Brzozowski, J. A. (1962). Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, *12*(6), 529—561.

Cabasino, M. P., Giua, A., Pocci, M., & Seatzu, C. (2011). Discrete-event diagnosis using labeled Petri nets. An application to manufacturing systems. *Control Engineering Practice*, *19*(9), 989—1001.

Cabasino, M. P., Giua, A., Paoli, A., & Seatzu, C. (2013). Decentralized diagnosis of discrete-event systems using labeled Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(6), 1477—1485.

Cai, K., & Wonham, W. M. (2010a). Supervisor localization: A top-down approach to distributed control of discrete-event systems. *IEEE Transactions on Automatic Control*, *55*(3), 605—618.

Cai, K., & Wonham, W. M. (2010b). Supervisor localization for large discrete-event systems. *The International Journal of Advanced Manufacturing Technology*, *50*(9), 1189—1202.

Čapkovič, F. (2017, March). Petri nets in discrete-event and hybrid systems modelling, analysing, performance evaluation and control. In R. Szewczyk, C. Zieliński & M. Kaliczyńska (Eds.), *Automation 2017. Innovations in automation, robotics and measurement techniques* (pp. 3—21). Springer.

Carvajal Soto, J. A., Tavakolizadeh, F., & Gyulai, D. (2019). An online machine learning framework for early detection of product failures in an industry 4.0 context. *International Journal of Computer Integrated Manufacturing, 32*(4-5), 452—465.

Cassandras, C. G., & Lafortune, S. (2009). *Introduction to discrete event systems*. Springer.

Chakraborti, T., Isahagian, V., Khalaf, R., Khazaeni, Y., Muthusamy, V., Rizk, Y., & Unuvar, M. (2020). From robotic process automation to intelligent process automation. In A. Asatiani, J. M. García, N. Helander, A. Jiménez-Ramírez, A. Koschmider, J. Mendling, G. Meroni, & H. A. Reijers (Eds.), *Business process management: Blockchain and robotic process automation forum* (pp. 215—228). Springer.

Chakraborty, S. (2003). *Formal languages and automata theory - Regular expressions and finite automata*. [White paper]. Swiss Federal Institute of Technology (ETH Zürich).

Chanthery, E., & Pencolé, Y. (2009). Monitoring and active diagnosis for discrete-event systems. *IFAC Proceedings Volumes*, *42*(8), 1545—1550.

Chaouiya, C. (2007). Petri net modelling of biological networks. *Briefings in Bioinformatics*, *8*(4), 210—219.

Cheng, J., Chen, W., Tao, F., & Lin, C. L. (2018). Industrial IoT in 5G environment towards smart manufacturing. *Journal of Industrial Information Integration, 10,* 10—19.

Cisco Systems, Inc. (2020, September 16). *Networking and security in industrial*

*automation environments*.

Combacau, M., Berruet, P., Zamai, E., Charbonnaud, P., & Khatab, A. (2000). Supervision and monitoring of production systems. *IFAC Proceedings Volumes*, *33*(17), 849—854.

Copeland, B. J. (Ed.). (2004). *The essential Turing*. Clarendon Press.

Crelier, A. (2019). *The challenges of scaling the Internet of Things* [White paper]. Center for Security Studies (CSS), ETH Zürich.

Dagkakis, G., & Heavey, C. (2015). A review of open source discrete event simulation software for operations research. *Journal of Simulation*, *10*(3), 1—14.

Darabi, H., Jafari, M. A., & Manapure, S. S. (2003). Finite automata decomposition for flexible manufacturing systems control and scheduling. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *33*(2), 168—175.

David, R., & Alla, H. (2010). *Discrete, continuous, and hybrid Petri nets*. Springer.

Davidrajuh, R. (2012). Designing Fault-Tolerant Autonomous Systems with Adaptive Supervisory Control. *Proceedings of the Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation*, 185—190. IEEE.

Descreye Solutions (2018, February 3). A comparison of discrete event simulation software. *Descreye*.  https://www.descreye.com/blog/a-comparison-of-discrete-event-simulation-software/

Desel, J., & Reisig, W. (1996). *Advanced course on Petri nets* (pp. 122—173). Springer.

DiCesare, F., Harhalakis, G., Proth, J. M., Silva, M., & Vernadat, F. B. (1993). *Practice of Petri nets in manufacturing*. Springer.

Dickman, F. (2009). *Hacking the industrial network.* Phoenix Contact.

Dietrich, P., Malik, R., Wonham, W. M., & Brandin, B. A. (2002). *Synthesis and control of discrete event systems* (pp. 185—201). Springer.

Dugyala, R., Reddy, N. H., & Kumar, S. (2019). Implementation of SCADA through cloud based IoT devices - Initial design steps. *Proceedings of the* *Fifth International Conference on Image Information Processing (ICIIP),* 367—372. IEEE. doi:10.1109/iciip47207.2019.8985966

Durakbasa, N. M., & Gençyılmaz, M. G. (2021). *Digital conversion on the way to Industry 4.0*. Springer.

Dynkin, E. B. (2012). *Theory of Markov processes*. Courier Corporation.

Ebbinghaus, H. D., Flum, J., & Thomas, W. (2013). *Mathematical logic*. Springer.

Elhabashya, A. E., Wells, L. J., & Camelio, J. A. (2019). Cyber-physical security research efforts in manufacturing: A literature review. *Procedia Manufacturing*, *34*, 921—931.

Ehlers, R., Lafortune, S., Tripakis, S., & Vardi, M. (2013). *Reactive synthesis vs. supervisory control: Bridging the gap* [Technical report no. UCB/EECS-2013-162]. EECS Department, University of California.

Erlang, A. K. (1909). The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B*, *20*, 33—39.

Feng, L., & Wonham, W. M. (2008). Supervisory control architecture for discrete-event systems. *IEEE Transactions on Automatic Control*, *53*(6), 1449—1461. DOI: [10.1109/TAC.2008.927679](https://www.researchgate.net/deref/http%3A%2F%2Fdx.doi.org%2F10.1109%2FTAC.2008.927679)

Figueiredo, J., & da Costa, J. S. (2012). A SCADA system for energy management in intelligent buildings. *Energy and Buildings*, *49*, 85—98.

Firouzi, F., Farahani, B., Weinberger, M., DePace, G., & Aliee, F. S. (2020). *Intelligent Internet of Things* (pp. 3—50). Springer.

Fomundam, S., & Herrmann, J. W. (2007). *A survey of queuing theory applications in healthcare* [Technical report no. 2007-04]. Institute for Systems Research, University of Maryland.

Gebali, F. (2015). Markov chains. In *Analysis of computer networks* (pp. 67—127). Springer.

Ghazel, M. (2011). Monitoring and diagnosis of discrete event systems using time Petri nets: A railway case study. In L. M. Simon (Ed.), *Fault detection: Theory, methods and systems* (pp. 69—95). Nova Science Publishers.

Giallanza, A., Aiello, G., Marannano, G., & Nigrelli, V. (2020). Industry 4.0: Smart test bench for shipbuilding industry. *International Journal on Interactive Design and Manufacturing*, *14*(4), 1525—1533.

Giambene, G. (2005). *Queuing theory and telecommunications*. Springer.

Giua, A., & Silva, M. (2017). Modeling, analysis and control of discrete event systems: A Petri net perspective. *IFAC-PapersOnLine*, *50*(1), 1772—1783.

Gold, R. (2004). *Petri nets in software engineering* [Working paper no. 1612-6483]. Fachhochschule Ingolstadt.

Goldston, J. L. (2019a). A qualitative study of risk in enterprise resource planning implementations. *Global Scientific Journal, 7*(12), 1129—1159.

Goldston, J. L. (2019b). Leadership approaches during digital transformations in small and medium enterprises. *IEEE - Science, Engineering, and Management, 7*(12), 89—106.

Goldston, J. L. (2019c). Critical success factors in small and medium enterprise ERP implementations. *International Journal of Scientific & Engineering Research, 10*(12), 1480—1533.

Goldston, J. L. (2020). The evolution of digital transformations: A literature review. *International Journal of Innovative Science and Research Technology, 5*(4), 9—18.

Goodwin, G. C., Graebe, S. F., & Salgado, M. E. (2001). *Control system design*. Prentice Hall.

Gu, C., Wang, X., & Li, Z. (2018). Synthesis of supervisory control with partial observation on normal state-tree structures. *IEEE Transactions on Automation Science and Engineering*, *16*(2), 984—997.

Gunes, V., Peter, S., Givargis, T., & Vahid, F. (2014). A survey on concepts, applications, and challenges in cyber-physical systems. *KSII Transactions on Internet & Information Systems, 8*(12), 4242—4268.

Gurrapu, S. (2017). *Connected sensors in industrial automation.* Texas Instruments.

Gyapay, S., Pataricza, A., Sziray, J., & Friedler, F. (2002). Petri net-based optimization of production systems. In W. Elmenreich, J. A. Tenreiro Machado, & I. J. Rudas (Eds.), Intelligent systems at the service of mankind (pp. 157—168). U Books.

Harrison, P. G., & Patel, N. M. (1992). *Performance modelling of communication networks and computer architecture*. Addison-Wesley Longman.

Heiner, M., Gilbert, D., & Donaldson, R. (2008). Petri nets for systems and synthetic biology. In M. Bernardo, P. Degano, & G. Zavattaro (Eds.), Formal methods for computational system biology (pp. 215—264). Springer.

Hofer, F. (2018). Architecture, technologies and challenges for cyber-physical systems in Industry 4.0: A systematic mapping study. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1—10. doi:10.1145/3239235.3239242

Hollender, M. (2010). *Collaborative process automation systems*. ISA.

Holmes, J. F., Russell, G., & Allen, J. K. (2013). Supervisory control and data acquisition (SCADA) and related systems for automated process control in the food industry: An introduction. In D. G. Caldwell (Ed.), *Robotics and automation in the food industry* (pp. 130—142). Woodhead Publishing.

Holusha, J. (1982, November 14). General Motors. A giant in transition. *The* *New York Times.*

Hopcroft, J. (1971). An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi & A. Paz (Eds.), *Theory of machines and computations* (pp. 189—196). Academic Press.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *ACM SIGACT News*, *32*(1), 60—65.

Humoreanu, B., & Nascu, I. (2012). Wastewater treatment plant SCADA application. *Proceedings of 2012 IEEE International Conference on Automation, Quality and Testing, Robotics*, 575—580. IEEE.

Hunter, T. (2020). The Chomsky hierarchy. In N. Allott, T. Lohndal & G. Rey (2020). *Blackwell Companion to Chomsky*. Blackwell Publishers.

Idghamishi, A. M., & Zad, S. H. (2004). Fault diagnosis in hierarchical discrete-event systems. *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC) Vol.1,* 63—68. IEEE.

Irannejad, M., & Iraninejad, M. (2014). Remote monitoring of oil pipelines cathodic protection system via GSM and its application to SCADA system. *International Journal of Science & Research*, *3*(5), 1619—1622.

ISA Global Security Alliance. (2020). Notable cyberattacks impacting IACS. *Security of Industrial Automation and Control Systems*.

Jensen, K. (1987). Coloured petri nets. In W. Brauer, W. Reisig & G. Rozenberg (Eds.). *Petri nets: Central models and their properties* (pp. 248—299). Springer.

Kaymakci, O., & Kurtulan, S. (2009). A novel performance evaluation method for DES. *Journal of Information Science & Engineering*, *25*(1), 105—120.

Kendall, D. G. (1953). Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*, *24*(3), 338—354.

Kobetski, A., & Fabian, M. (2009). Time-optimal coordination of flexible manufacturing systems using deterministic finite automata and mixed integer linear programming. *Discrete Event Dynamic Systems*, *19*(3), 287—315.

Košecká, J. (1992). *Control of discrete event systems* [Technical report no. MS-CIS-92-35]. Department of Computer and Information Science, University of Pennsylvania.

Kuroda, S. Y. (1964). Classes of languages and linear-bounded automata. *Information and Control*, *7*(2), 207—223.

Langmann, R., & Stiller, M. (2019). The PLC as a smart service in Industry 4.0 production systems. *Applied Sciences, 9*(18), 1—20.

Lawson, M. V. (2003). *Finite automata*. CRC Press.

Leander, B., Čaušević, A., & Hansson, H. (2019). Applicability of the IEC 62443 standard in Industry 4.0/IIoT. *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 1—8. ACM.

Little, J. D., & Graves, S. C. (2008). Little's law. In D. Chhajed & T. J. Lowe (Eds.), *Building intuition: Insights from basic operations management models and principles* (pp. 81—100). Springer.

Lipsky, L. (2009). M/M/1 queue. In *Queueing theory – A linear algebraic approach* (pp. 33—75). Springer. https://doi.org/10.1007/978-0-387-49706-8\_2

Lohmann, N., Verbeek, E., & Dijkman, R. (2009). Petri net transformations for business processes—A survey. In W. M. P. van der Aalst & K. Jensen (Eds.), *Transactions on Petri nets and other models of concurrency II* (pp. 46—63). Springer.

Lu, Y., Riddick, F., & Ivezic, N. (2016). The paradigm shift in smart manufacturing system architecture. In I. Nääs, O. Vendrametto, J. M. Reis, R. F. Gonçalves, M. T. Silva, G. von Cieminski, & D. Kirisitis (Eds.), *Advances in production management systems* (pp. 767—776)*.* Springer.

Macías, E. J., & de la Parte, M. P. (2004). Simulation and optimization of logistic and production systems using discrete and continuous Petri nets. *Simulation*, *80*(3), 143—152.

Marsan, M. A. (1988). Stochastic Petri nets: An elementary introduction. In G. Rozenberg (Ed.), *Advances in Petri nets 1989* (pp. 1—29). Springer.

Marsudi, M., & Shafeek, H. (2014). The application of queuing theory in multi-stage production line. *Proceedings of the 2014 International Conference on Industrial Engineering and Operations Management*, 668—675.

Masek, J., Camaj, J., & Nedeliakova, E. (2015). Application the queuing theory in the warehouse optimization. *International Journal of Industrial and Manufacturing Engineering, 9*(11)*,* 3744—3748*.*

Matloff, N. (2008). *Introduction to discrete-event simulation and the SimPy language*. [Doctoral dissertation, University of California] ResearchGate.

Medida, S. (2008). *Industrial automation pocket guide*. IDC Technologies.

Meena, D. (2019, October 10). How AI is making preventive maintenance a reality for the manufacturing industry. *Pluto7*.

Mehandiratta, R. (2011). Applications of queuing theory in health care. *International Journal of Computing and Business Research*, *2*(2), 2229—6166.

Merlin, P. (1974). *A study of the recoverability of computer systems* [Doctoral dissertation, University of California]. University of California.

Moghaddam, M., Cadavid, M. N., Kenley, C. R., & Deshmukh, A. V. (2018). Reference architectures for smart manufacturing: A critical review. *Journal of Manufacturing Systems, 49*(1), 215—225.

Moore, E. F. (1956). Gedanken-experiments on sequential machines. In C. E. Shannon & J. McCarthy (Eds.), *Automata studies* *(AM-34)* (pp. 129—153). Princeton University Press.

Morgenstern, A., & Schneider, K. (2007). Synthesizing deterministic controllers in supervisory control. In J. Filipe, J. L. Ferrier, J. A. Cetto, & M. Carvalho (Eds.), *Informatics in control, automation and robotics II* (pp. 95—102). Springer.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541—580.

National Science and Technology Council. (2020). *Artificial intelligence and cybersecurity: Opportunities and challenges*. The Networking and Information Technology Research and Development Program.

Nerode, A. (1958). Linear automaton transformations. *Proceedings of the American Mathematical Society*, *9*(4), 541—544.

Olsen, T. L., & Tomlin, B. (2020). Industry 4.0: Opportunities and challenges for operations management. *Manufacturing & Service Operations Management, 22*(1), 113—122.

Panek, S., Stursberg, O., & Engell, S. (2006). Efficient synthesis of production schedules by optimization of timed automata. *Control Engineering Practice*, *14*(10), 1183—1197.

Parth, S. (2020, February 4). How confidence and prediction intervals work. *Towards Data Science*.

Pawlewski, P. (2012). Petri Nets. Manufacturing and Computer Science. IntechOpen.

Pencolé, Y., Kamenetsky, D., & Schumann, A. (2006). Towards low-cost fault diagnosis in large component-based systems. *IFAC Proceedings Volumes*, *39*(13), 1473—1478.

Peterson, J. L. (1977). Petri nets. *Computing Surveys*, *9*(3), 223—252.

Piera, M. À., Narciso, M., Guasch, A., & Riera, D. (2004). Optimization of logistic and manufacturing systems through simulation: A colored Petri net-based methodology. *Simulation*, *80*(3), 121—129.

Pinheiro, L. P., Lopes, Y. K., Leal, A. B., & Rosso, R. S. U. J. (2015). Nadzoru: A software tool for supervisory control of discrete event systems. *IFAC-PapersOnLine*, *48*(7), 182—187.

Popova-Zeugmann, L. (2013). Time Petri nets. In *Time and Petri nets (*pp. 31—137). Springer.

Rabin, M. O., & Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, *3*(2), 114—125.

Rajeswari, V., Suresh, L. P., & Rajeshwari, Y. (2013, March). Water storage and distribution system for pharmaceuticals using PLC and SCADA. *Proceedings of the 2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*, 79—86. IEEE.

Ramadge, P. J., & Wonham, W. M. (1987). Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, *25*(1), 206—230.

Ramchandani, C. (1974). *Analysis of asynchronous concurrent systems by Petri nets* [Doctoral dissertation, MIT]. Massachusetts Institute of Technology.

Ran, N., Wang, S., Su, H., & Wang, C. (2017). Fault diagnosis for discrete event systems modeled by bounded Petri nets. *Asian Journal of Control*, *19*(4), 1532—1541.

Rashid, R., Hoseini, S. F., Gholamian, M. R., & Feizabadi, M. (2015). Application of queuing theory in production-inventory optimization. *Journal of Industrial Engineering International*, *11*(4), 485—494.

Ray, S., Miers, D., Tornbohm, C., & Kerremans, M. (2019). *Move beyond RPA to deliver hyperautomation*. Gartner.

Razzhivina, M. A., Yakimovich, B. A., & Korshunov, A. I. (2015). Application of information technologies and principles of lean production for efficiency improvement of machine building enterprises. *Pollack Periodica, 10*(2), 17—23.

Reiser, C., Da Cunha, A. E., & Cury, J. E. (2006). The environment grail for supervisory control of discrete event systems. *Proceedings of the 2006 8th International Workshop on Discrete Event Systems*, 390—391. IEEE.

Reisig, W. (2012). *A primer in Petri net design*. Springer.

Reveliotis, S., & Nazeem, A. (2014). Deadlock avoidance policies for automated manufacturing systems using finite state automata. In J. Campos, C. Seatzu, & X. Xie (Eds.), *Formal methods in manufacturing* (pp. 169—195). CRC Press.

Révész, G. E. (1991). *Introduction to formal languages*. Courier Corporation.

Ricker, L., Lafortune, S., & Genc, S. (2006, July). Desuma: A tool integrating giddes and umdes. *Proceedings of the 2006 8th International Workshop on Discrete Event Systems*, 392—393. IEEE.

Rio, E. (2017). *Asymptotic theory of weakly dependent random processes* (Vol. 80). Springer.

Robles, T., Alcarria, R., Martín, D., Navarro, M., Calero, R., Iglesias, S., & López, M. (2015). An IoT based reference architecture for smart water management processes. *JoWUA, 6*(1), 4—23.

Ru, Y., & Hadjicostis, C. N. (2009). Fault diagnosis in discrete event systems modeled by partially observed Petri nets. *Discrete Event Dynamic Systems,* *19*(4).

Rudie, K. (2006, July). The integrated discrete-event systems tool. *Proceedings of the 2006 8th International Workshop on Discrete Event Systems*, 394—395. IEEE.

Salomaa, A. (2014). *Theory of automata*. Pergamon.

Schumacher, A., Erol, S., & Sihn, W. (2016). A maturity model for assessing Industry 4.0 readiness and maturity of manufacturing enterprises. *Procedia CIRP*, *52*(1), 161—166.

Seatzu, C., Silva, M., & Van Schuppen, J. H. (2013). *Control of discrete-event systems.* Springer.

Sheridan, T. B. (1976). Toward a general model of supervisory control. In T. B. Sheridan & G. Johannsen (Eds.). *Monitoring behavior and supervisory control* (pp. 271—281). Springer.

Shiomi, M., Kanda, T., Nohara, K., Ishiguro, H., & Hagita, N. (2009). Adaptive supervisory control of a communication robot that approaches visitors. In H. Asama, H. Kurokawa, J. Ota, & K. Sekiyama (Eds.), *Distributed autonomous robotic systems 8* (pp. 555—564). Springer.

Sigelle, M., Jermyn, I., & Perreau, S. (2009). *Markov chains, diffusion and Green functions. Applications to traffic routing in ad hoc networks and to image restoration*. Telecom ParisTech.

Spiceworks, Inc. (2020). *The 2020 state of IT.* Available online.

Srimani, P. K., & Nasir, S. F. B. (2007). *A textbook on automata theory*. Foundation Books.

Starke, P. H. (1990). *Analyse von Petri-Netz-Modellen* [*Analysis of Petri net models*]. Springer.

Strauss, C. (2003). *Practical electrical network automation and communication systems*. Elsevier.

Stroock, D. W. (2013). An introduction to Markov processes. Springer.

Su, R., & Wonham, W. M. (2006). Hierarchical fault diagnosis for discrete-event systems under global consistency. *Discrete Event Dynamic Systems*, *16*(1), 39—70.

Swamidass, P. M. (Ed.). (2000). *Encyclopedia of production and manufacturing management*. Springer.

Thalheim, B. (2009). Extended entity-relationship model. In Liu, L., & Öszu, M. T. (Eds.), *Encyclopedia of database systems* (pp. 1083—1091). Springer.

Thombansen, U., Buchholz, G., Frank, D., Heinisch, J., Kemper, M., Pullen, T., Reimer, V., Rotshteyn, G., Schwenzer, M., Stemmler, S., Abel, D., Gries, T., Hopmann, C., Klocke, F., Poprawe, R., Reisgen, U., & Schmitt, R. (2018). Design framework for model-based self-optimizing manufacturing systems. *The International Journal of Advanced Manufacturing Technology*, *97*(1-4), 519—528.

Trunzer, E., Calà, A., Leitão, P., Gepp, M., Kinghorst, J., Lüder, A., Schauerte, H., Reifferscheid, M., & Vogel-Heuser, B. (2019). System architectures for Industrie 4.0 applications. *Production Engineering, 13*(3-4), 247—257.

Van der Aalst, W. M. P. (2002). Making work flow: On the application of Petri nets to business process management. In J. Esparza & C. Lakos (Eds.), *Application and theory of Petri nets 2002* (pp. 1—22). Springer.

van der Aalst, W. M. P. (2019). Everything you always wanted to know about Petri nets, but were afraid to ask. In T. Hildebrandt, B. F. van Dongen, M. Röglinger & J. Mendling (Eds.), *Business Process Management* (pp. 3—9). Springer.

Van Kampen, N. G. (1992). *Stochastic processes in physics and chemistry*. Elsevier.

van Putten, B. J. C., van der Sanden, B., Reniers, M., Voeten, J., & Schiffelers, R. (2020). Supervisor synthesis and throughput optimization of partially-controllable manufacturing systems. *Discrete Event Dynamic Systems,* *30*(4), 1—33.

Vilaplana, J., Solsona, F., Teixidó, I., Mateo, J., Abella, F., & Rius, J. (2014). A queuing theory model for cloud computing. *The Journal of Supercomputing*, *69*(1), 492—507.

Vossen, G., & Witt, K. U. (2000). *Grundlagen der theoretischen Informatik mit Anwendungen* [*Fundamentals of theoretical computer science with applications*]. Vieweg-Verlag.

Walter, G. G., & Contreras, M. (2012). *Compartmental modeling with networks*. Springer.

Wang, X., & Ray, A. (2004). A language measure for performance evaluation of discrete-event supervisory control systems. *Applied Mathematical Modelling*, *28*(9), 817—833.

Weidele, M. (2018, February 07). *IEC 62443 – Diese Grundlagen sollten Sie als Betreiber einer Automatisierungslösung kennen* [*IEC 62443 – Basics you should know as the operator of an automation solution*]. Sichere Industrie.

Wonham, W. M., & Cai, K. (2019). *Supervisory control of discrete-event systems*. Springer.

Xing, K., Zhou, M., Liu, H., & Tian, F. (2008). Optimal Petri-net-based polynomial-complexity deadlock-avoidance policies for automated manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, *39*(1), 188—199.

Xu, L., Li, J., Hua, J., Li, X., & Ouyang, M. (2009). Adaptive supervisory control strategy of a fuel cell/battery-powered city bus. *Journal of Power Sources*, *194*(1), 360—368.

Ye, X., Zhou, J., & Song, X. (2003). On reachability graphs of Petri nets. *Computers & Electrical Engineering*, *29*(2), 263—272.

Yu, C. C. (2006). *Autotuning of PID controllers: A relay feedback approach*. Springer.

Żak, J., & Jacyna-Gołda, I. (2013). Using queue theory to analysis and evaluation of the logistics centre workload. *Archives of Transport*, *25-26*(1-2), 117—135.

Zamaï, E., Combacau, M., & Subias, A. (1998). Models and strategies for monitoring of flexible manufacturing systems. *IFAC Proceedings Volumes,* *31*(15), 903—908.

Zamfirescu, C. B., Pirvu, B. C., Loskyll, M., & Zuehlke, D. (2014). Do not cancel my race with cyber-physical systems. *IFAC Proceedings, 47*(3), 4346—4351.

Zaytoon, J., & Lafortune, S. (2013). Overview of fault diagnosis methods for discrete event systems. *Annual Reviews in Control*, *37*(2), 308—320.

Zeid, A., Sundaram, S., Moghaddam, M., Kamarthi, S., & Tucker, M. (2019). Interoperability in smart manufacturing: Research challenges*. Machines, 7*(21), 1—17.

Zhang, S., Wu, N., Li, Z., Qu, T., & Li, C. (2017). Petri net-based approach to short-term scheduling of crude oil operations with less tank requirement. *Information Sciences*, *417*, 247—261.

Zuberek, W. M. (1991). Timed Petri nets definitions, properties, and applications. *Microelectronics Reliability,* *31*(4), 627—644.