Institut für Intermodale Transport- und Logistiksysteme

Technische Universität Braunschweig

Simulation of mechatronic systems (Simulation mechatronischer Systeme)

Lecture Notes

Jürgen Pannek

April 4, 2024

Jürgen Pannek
Institute for Intermodal Transport and Logistic Systems
Hermann-Blenck-Str. 42
38519 Braunschweig

FOREWORD

During summer term 2024 I give the lecture to the module *Simulation of mechatronic systems (Simulation mechatronischer Systeme)* at the Technical University of Braunschweig. To structure the lecture and support my students in their learning process, I prepared these lecture notes. The notes are work in progress and are updated in due course of the lecture itself. Moreover, I will integrate remarks and corrections throughout the term.
The aim of the module is to classify simulation techniques from numerical mathematics and apply these to mechatronic case studies. After completing the module, the students shall be able to recall, categorize, apply, select and rate simulation methods to mechatronic use cases. Moreover, students shall be able to describe, explain, evaluate, analyze and assess simulation results. As such, students shall be capable to derive and apply automation procedures for deployment, simulation and testing of digital models.
To this end, we will tackle the subject areas

- dynamical systems,
- software development,
- simulation and testing,
- visualization, and
- process automation

within the lecture. To support students, we utilize university resources such as GitLab to assist students to create a professional software development tool chain and interact with cluster computing technology within the tutorial classes. The module itself is accredited with 5 credits.
An electronic version of this script can be found at

https://www.tu-braunschweig.de/en/itl/teaching/lecture-notes

Literature for further reading

- Dynamical systems
  - DEUFLHARD, P. ; BORNEMANN, F.: *Scientific computing with ordinary differential equations*. Springer, 2012
  - KHALIL, H.K.: *Nonlinear Systems*. Prentice Hall, 2002
  - BULIRSCH, R. ; STOER, J.: *Numerische Mathematik 2*. Springer, 2005
- Software development
  - CHACON, S. ; STRAUB, B.: *Pro Git*. Apress, 2022
  - FARLEY, D.: *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, 2022
  - IGLBERGER, K.: *C++ Software Design: Design Principles and Patterns for High-Quality Software*. O'Reilly Media, 2022
- Simulation and testing
  - GLÖCKLER, M.: *Simulation mechatronischer Systeme: Grundlagen und technische Anwendung*. Springer, 2014
  - ZIEGLER, B.P. ; MUZY, A. ; KOFMAN, E.: *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Academic press, 2018
- Visualization
  - WILKE, C.O.: *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media, 2019
- Process automation
  - COWELL, C. ; LOTZ, N. ; TIMERLAKE, C.: *Automating DevOps with GitLab CI/CD Pipelines*. Packt Publishing, 2023
  - LUNZE, J.: *Automatisierungstechnik*. 5th edition. DeGruyter, 2020

Contents

List of Tables

List of Figures

List of Definitions and Theorems

LIST OF ALGORITHMS

CHAPTER 1 INTENTION

Within this introduction chapter we strive to lay the foundations for the overall lecture. As the above quote already indicates, simply doing a simulation without having a proper theory on how to analyze the results and design the experiments is just number crunching.
Within Section 1.1, we introduce the V model for development and discuss the purpose of simulation in different types of scenarios. The following Section 1.2 introduces the concept of DevOps and system architecture in software development, which we will cover in the tutorials in parallel to the lecture. Last, we shortly address the deployment on computing clusters using Kubernetes and Docker in Section 1.3.

1.1 Aim of lecture

As outlined in the foreword, the lecture provides insights into dynamical systems, software development, simulation and testing, visualization, and process automation. To this end, we discuss both the overall methodology but also specific methods within the steps of the overall methodology. Based on VDI/VDE 2206 guideline [16], the lecture itself is driven by the idea of development for mechatronic and cyber-physical systems as it is currently found in basically any company, cf. Figure 1.1.
Note that the V model representation in Figure 1.1 already includes the cycles that are found in the so called agile approach. To learn these techniques, in the tutorial classes we utilize a state-of-the-art toolchain to implement, integrate, test and automate our example applications.
Taking one step back, we must first ask ourselves what the aim of simulation actually is. On a very fundamental level, there are three different types of scenarios for which simulation can be used in applications:

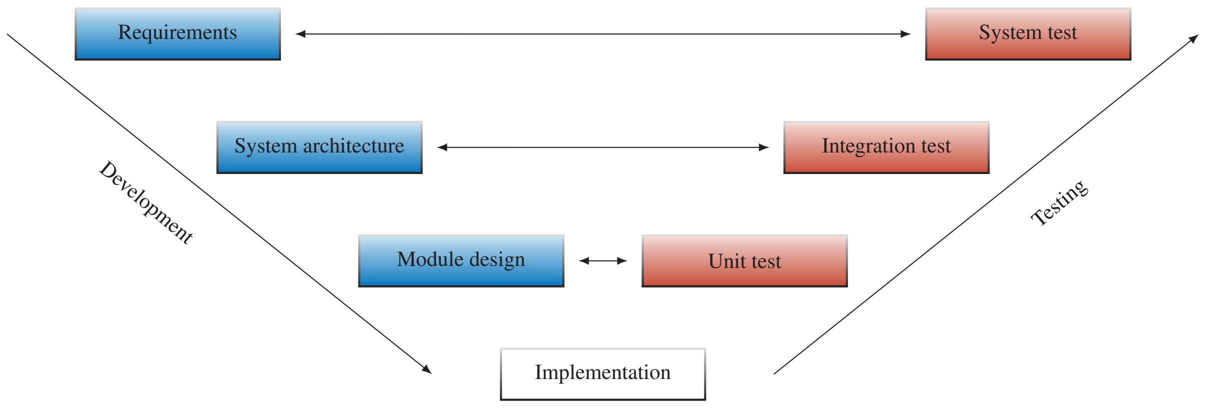


Figure 1.1: V model for system development

1. Understand a known scenario:
   In this setting, the scenario itself is both known and real data is available. Examples may range from driving/drifting behavior of cars, reaction of mechanical components to software behavior or accidents. Here, the aim is to gather insight into the scenario and understand connections, i.e. the question "why does something happen the way it happens".

2. Optimize a known scenario:
   Building up on the previous case, optimization aims to generate a wanted behavior. In this case, typically no real data is available as testing is rather expensive. Typical examples are timetables for truck fleets, performance tests for engines, material tests, reaction of software in application scenarios and so forth. Hence, simulation is used to improve insight into possible improvements which require real time tests later, i.e. the question "how can something be utilized to improve given criteria".

3. Predict unknown scenario:
   Using the extrapolation idea from the previous case, the idea of using simulation in an unknown scenario is to predict possible behavior and quantify uncertainty of the results. In contrast to the previous case, no testing is possible. Examples are climate change, fine dust distribution prediction for cities, properties of new materials etc. Here, simulation is used to provide us with a range within which the truth most likely can be found, i.e. the question "is something likely to happen or to work out fine".

The lecture itself runs in parallel to the lecture *Modeling of mechatronic systems (Modellierung mechatronischer Systeme)*, within which models and modeling techniques are developed, which can be used for simulation. Within the tutorial, we deepen the understanding of simulation by using a full stack software development tool and optionally middleware for computing.

1.2 DevOps, Toolchain and System Architecture

On the implementation side of simulation, modern instruments can be used to utilize an agile/DevOps approach for implementation. The idea of the latter is to work in sprints to first create a working prototype with basic functionality, which is then extended/improved in following sprints. The typical picture to be seen in the literature is given in Figure 1.2.

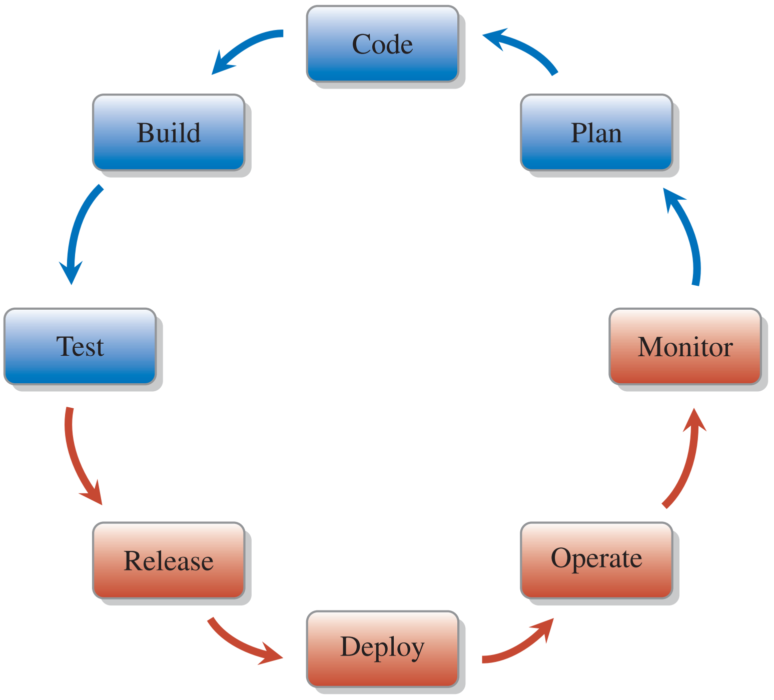


Figure 1.2: Agile development

Remark 1.1
We like to note that Figure 1.2 shows the extended version used in DevOps, which treats software already in use. In the prelaunch phase or in pre-/development, the operate and monitor parts are omitted.

As DevOps aims to significantly increase the applicability/usability of the software (and the satisfaction of customers), DevOps is an endless loop and in many cases also drawn as a lying infinity symbol.
Within the loop, the tasks correspond to the following:

- Plan: In this part, tasks are organized, set up and scheduled in a chosen management tool. The idea is to plan tasks using the user story process from the agile methodology. Writing tickets in the form of a user story will allow Devs and Ops to understand what development needs to be done and why. A perfect user story exhibits a what (who, where, trigger), a why, and acceptance criteria.

- Code: Within this part, Devs are doing code development and code review. When the code is ready, they merge it. In DevOps practice, it is important to share a code tool between Ops and developers teams.
- Build: In this step, the first step toward automation is done. The goal here is to build the source code into one desired format, compiling, testing, and deploying in a particular place of the infrastructure. With this setup, continuous integration (CI) and delivery (CD) tools can check and verify the source code from Source Code Management and build it.
- Test: Continuing the build task, continuous testing is applied to reduce risks of errors in the code. Automatic tests ensure that no bugs will be deployed to users. Therefore, testing tools need to be implemented in the workflow to ensure the quality of the software.
- Release: After being tested, the software is made ready for deployment. Within this step, the process is handed over from Dev to Ops.
- Deploy: In this step, the Ops deploy the new software upon user systems. Similar to CI/CD, a continuous deployment can be used for automation.
- Operate/ configure infrastructure: In this step, a scalable infrastructure is built and maintained by Ops. Additionally, security issues and log management are taken care of.
- Monitor: Last, monitoring concludes the circle and allows to identify issues and wishes, which are then fed to Devs in the planning step.

Within the tutorials of this lecture, we will consider a tool chain based on GitLab, which allows for code maintenance, issue tracking, CI/CD, automatic testing and automatic deployment. Further details will be given in the tutorial itself.
Over the last years, Git is the most prominent version control system due to its distributed architecture, flexibility, performance and branching/merging capabilities. GitLab and other packages are DevOps software stacks, that can be used to develop, secure and operate applications. Figure 1.3 sketches the steps within a typical version control system.
Within Figure 1.3, the respective colors indicate branches, which allow for

- parallel development and
- structuring the development, test and release process.

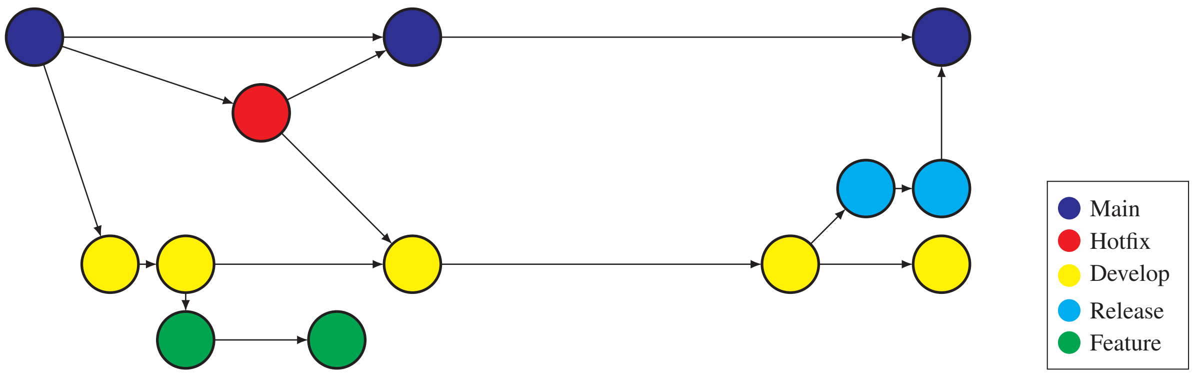


Figure 1.3: Version control including CI/CD pipline

In parallel development, each developer (typically) uses a separate development branch to implement new features or improve the code. If the development is done and successfully tested, it is merged into the main development branch. The process within the DevOps cycle, cf. Figure 1.2, is also managed in separate branches. Here, one typically uses branches for development, for releases, hotfixes and a main one.
The application as well as its testing is typically split into classes. These classes are defined once with fixed interfaces, but may be modified in their interior. This allows for modularity in usage, e.g., in different applications, as well as maintainability of the classes themselves. Figure 1.4 shows a diagram type typically used to visualize classes and their interaction.

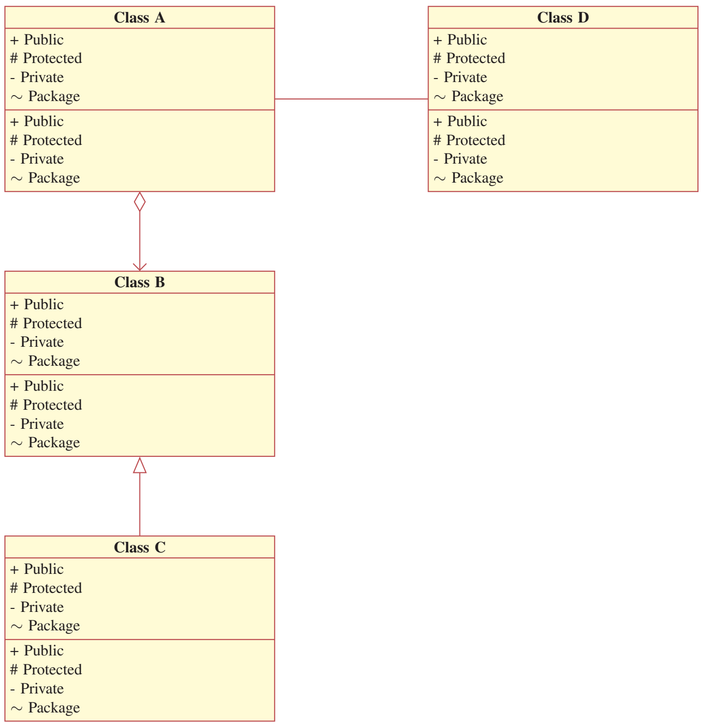


Figure 1.4: Sketch of UML diagram of classes

Within Figure 1.4, each class exhibits three components:

- Name
- Attributes
- Operations

Both attributes and operations may come in one of the four types

- public – can be accessed from the outside
- protected – can be accessed by associated classes
- private – cannot be accessed
- package – are linked to external

indicating their usability.
Using these classes, objects are derived as instances of the classes. Hence, several objects of the same class can exist in parallel using the same code but different names for the objects.
Moreover, the lines and symbols linking classes visualize their interaction. Here, we focus on the possibilities of

- aggregation (indicated between Class A and Class B) – one Class A object deals with possible several Class B objects
- inheritance (indicated between Class B and Class C) – Class C has all properties of Class B and can add new ones or modify existing one
- association (indicated between Class A and Class D) – one Class A object and one Class D object can access their protected attributes/operations

Connected to the software development process is the hardware computing side, which we touch in the following section.

1.3 Kubernetes and Docker

Upon usage of code, the applications need to be deployed onto a hardware for computation. While for learning how to program or for prototype applications this can be done straight forward by compiling and running the code, e.g., on a personal computer, the latter does not make sense for coordinated software development and especially not for large scale simulations.
To handle the latter, technologies such as Docker and Kubernetes can be used. Within the lecture and the tutorial, we will only scratch the surface and indicate how to use the latter. Yet, we aim to point out that the combination of both allows advantages over traditional and virtual deployment as indicated in Figure 1.5.

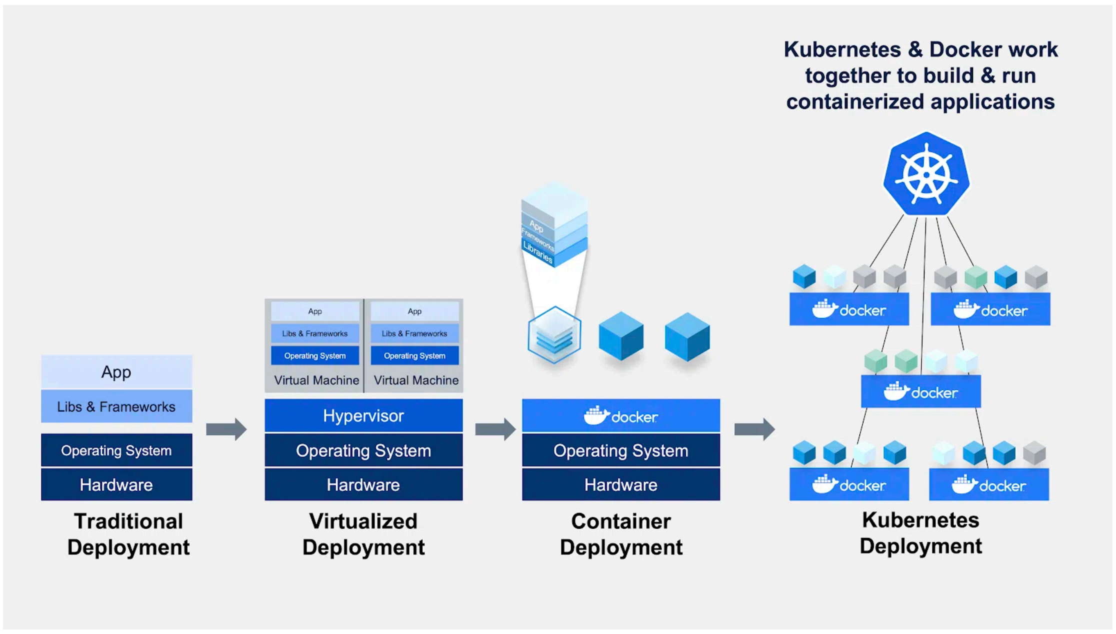


Figure 1.5: Tradition, virtualized and containerized deployment on distributed hardware[1]

Regarding the technologies themselves, Docker is a platform for developing, shipping, and running applications in containers. Here, the idea is to render an application to be standalone, i.e. to be portable between deployment systems and to be as lightweight as possible. Hence, such a container must include everything needed to run an application, including the code, runtime, libraries, and dependencies. Docker as a software package itself provides tools for building, managing, and deploying containers efficiently to any infrastructure that supports Docker.
Kubernetes, on the other hand, is a different infrastructure tool that allows to harmonize and run software across hardware. Roughly speaking, Kubernetes combines computer, which are then only a node and may be spread over a network, and makes them accessible like a single computer. Hence, it hides all the required communication and job assignment between computers. From the software development side, it automates the deployment, scaling, and management of containerized applications across clusters of nodes.
Combined, Kubernetes provides a highly scalable, resilient, and portable infrastructure for running containers. It abstracts away the underlying infrastructure, allowing developers to focus on building and deploying applications without worrying about the details of managing individual containers or nodes. As such, it is ideal for deploying and managing modern, cloud-native applications at scale.

In the following, we will focus on simulation from the theoretical and methodological point of view, and implement methods and examples using GitLab as a DevOps tool. While complementing modeling, we still require the respective terms which we introduce next.

[1]Source: https://www.docker.com

CHAPTER 2 BASICS FOR SYSTEMS AND SIMULATION

Theory provides the maps that turn an uncoordinated set of experiments or computer simulations into a cumulative exploration.

David E. Goldberg[1]

Within this chapter, we focus on terminology and fundamental description for systems and simulation. In particular, we formally define which kind of systems we will be working with, and how simulation is linked to such systems. To this end, we derive the representation of a system/process and of its model from basic norms in Section 2.1. In order to be directly usable, we formally define a system/process and its model in Section 2.2 using a standard mathematical formulation from the literature. In the subsequent Section 2.3, we characterize the term simulator, simulation and assessment, which will be used throughout the lecture. In the last Section 2.4, we introduce an example of a system model, which we will use to illustrate the steps of simulation in the upcoming chapters.

2.1 Systems

Within the lecture, we utilize definitions and common notation from the books of Lunze [14], Khalil [13] and Sontag [15].
We start with object we are actually interested in, that is the system or process at hand. Although used in various different scientific and non-scientific areas, the term is often not defined clearly but on a rather high level. In the literature [5], we see the following description for a system (translated from German):

[1] *1953, American computer scientist

> A system is a set of interrelated elements that are viewed as a whole in a particular context and considered as distinct from their environment.
>
> DIN IEC 60050-351 (2014)

Building on this description, a process is given as follows (translated from German):

> A process is the entirety of relations and interacting elements in a system through which matter, energy or information is transformed, transported or stored.
>
> DIN IEC 60050-351 (2014)

As we are interested in simulation, we have to be more specific and mathematically defined the following:

Definition 2.1 (System and process).
Consider two sets $\mathcal{U}$ and $\mathcal{Y}$. Then a map $\Sigma : \mathcal{U} \to \mathcal{Y}$ is called a *system* and the application of this map to an input $\mathbf{u} \in \mathcal{U}$ to obtain an output $\mathbf{y} = \Sigma(\mathbf{u}) \in \mathcal{Y}$ is called a *process*.

In particular, the sets $\mathcal{U}$ and $\mathcal{Y}$ are called input and output sets. An element from the input set $\mathbf{u} \in \mathcal{U}$ is called an input, which act from the environment to the system and is not dependent on the system itself or its properties, cf. Figure 2.1.

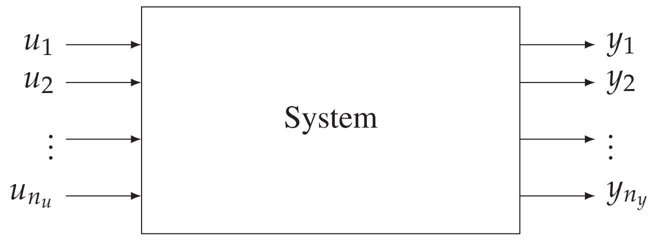


Figure 2.1: Term of a system

We distinguish between inputs, which are used to specifically manipulate (or control) the system, and inputs, which are not manipulated on purpose. We call the first ones *control or manipulation inputs*, and we refer to the second ones as *disturbance inputs*. An element from the output set $\mathbf{y} \in \mathcal{Y}$ is called an output. In contrast to an input, the output is generated by the system and influences the environment.

Remark 2.2
Note that in most cases not all measurable outputs are actually measured. Similarly, in many cases not all manipulable inputs are controlled.

To continue with a system, we need to introduce the concept of time:

Definition 2.3 (Time).
A *time set* $\mathcal{T}$ is a subgroup of $(\mathbb{R}, +)$.

Time allows us to let a system evolve. To see this point, we consider two electical systems illustrated in Figure 2.2 which represent an ideal resistance and an ideal capacitor.

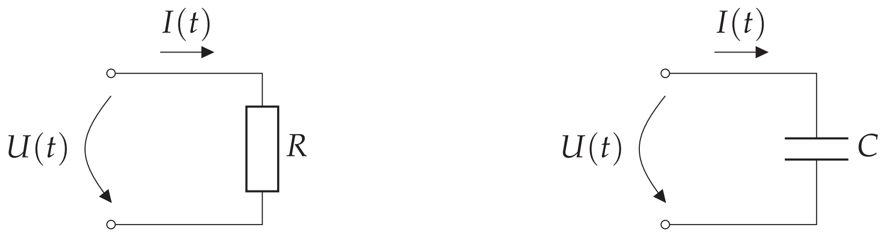


Figure 2.2: Example of static and dynamic systems

The systems in Figure 2.2 possess the input variable $I(t)$, the output variable $U(t)$ and time t. For the resistance R the output is uniquely defined by the input for every time instant t, i.e. we have

$$y(t) = U(t) = R \cdot I(t) = R \cdot u(t). \tag{2.1}$$

If the outputs depend on the input at the same time instant, we call systems such as this one *static*.

In contrast to this, the computation of the voltage $U(t)$ at the capacitor C at time instant t depends on the entire history $I(\tau)$ for $\tau \leq t$, i.e. we have

$$y(t) = U(t) = \frac{1}{C} \int\limits_{-\infty}^{t} I(\tau) d\tau = \frac{1}{C} \int\limits_{-\infty}^{t} u(\tau) d\tau.$$

If we additionally know the voltage $U(t_0)$ at a time instant $t_0 \leq t$, then only the history $t_0 \leq$

$\tau \leq t$ of the current is required, i.e.

$$y(t) = U(t) = \frac{1}{C} \int\limits_{-\infty}^{t} I(\tau) d\tau = \underbrace{\frac{1}{C} \int\limits_{-\infty}^{t_0} I(\tau) d\tau}_{U(t_0)} + \frac{1}{C} \int\limits_{t_0}^{t} I(\tau) d\tau = U(t_0) + \frac{1}{C} \int\limits_{t_0}^{t} u(\tau) d\tau. \quad (2.2)$$

As we can see from (2.1), the initial value $U(t_0)$ contains all the information on the history $\tau \leq t_0$. For this reason, one typically refers to $U(t_0)$ as the internal *state* of the system capacitor at time instant t_0. If the output of the system depends not only on the input at the time instant but also on the history of the latter, we call these systems *dynamic*.

Remark 2.4
Note that by this definition the set of dynamic systems covers the set of static systems.

If for a system according to Figure 2.1 the outputs $y_1(t), \ldots, y_{n_y}(t)$ depend on the history of the inputs $u_1(\tau), \ldots, u_{n_u}(\tau)$ for $\tau \leq t$ only, then the system is called *causal*. As all technically feasible systems are causal, we will restrict ourselves to this case.

Our discussion so far allow us to give the general definition of states of a dynamical system:

Definition 2.5 (State).
Consider a system $f : \mathcal{U} \to \mathcal{Y}$. If the output $\mathbf{y}(t)$ uniquely depends on the history of inputs $\mathbf{u}(\tau)$ for $t_0 \leq \tau \leq t$ and some $\mathbf{x}(t_0)$, then the variable $\mathbf{x}(t)$ is called *state* of the system.

Link: For further details on how to design inputs/controls such that system properties regarding outputs can be generated, we refer to the lectures *Control Engineering 1 & 2*.

In contrast to the input/output description of a system in Definition 2.1, states allow us to look inside a system. In engineering practice, the pure input/output description is also called black box and the state description as white box. The states and their development over time allow us to deduce and interpret properties such as long term behavior (e.g. convergence or stability) and short term one (e.g. swing-overs or transients), which can be used to describe the system by these properties.

Task 2.6
Which variable represents a state in case of induction?

Solution to Task 2.6: Current through the inductor

In order to understand and utilize a system or process, we require a model of it. Technically, the mathematical description we gave in Definition 2.1 is already a model. Yet in engineering practice, the term model refers to a partial representation of a system/process

2.2 Models and modeling

Abstracting from a system, a model represents those properties/parts which are of interest to the user of the model. Here, we define the following:

Definition 2.7 (Model).
Consider a system Σ with input and output sets $\mathcal{U}$ and $\mathcal{Y}$. Then a map $\Sigma_M : \mathcal{U} \to \mathcal{Y}$ is called a *model* of the system Σ if $\Sigma_M \sim \Sigma$.

So if we use a model, there may be deviations between model prediction and reality, both on long and short time horizons depending on the choice of the model. As a model does not describe all of reality, the system/process is split into two parts,

- the model, which describes what we are interested in, and
- the remainder, which contains everything else.

Since we cannot tell anything about the remainder (as it is not modeled), interactions between model and remainder can only be interpreted as disturbances.

Remark 2.8
Note that a disturbance for a system/system refers to unknown influences (e.g. environment) whereas a disturbance for a model indicates unmodeled influences (e.g. details of the system/process or environment).

Link: For further details on modeling, we refer to the lecture *Modeling of mechatronic systems* and for process modeling, identification and handling disturbances, we refer to the lecture *Systemics*.

For the models we consider in simulation, we assume the following to be satisfied:

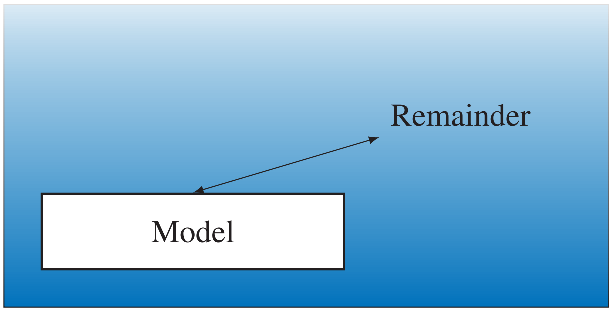


Figure 2.3: Model and remainder

Assumption 2.9 (Principles of modeling)
During the modeling process, six principles need to be met:

1. Principle of Correctness: A model needs to present the facts correctly regarding structure and dynamics (semantics). Specific notation rules have to be considered (syntax).

2. Principle of Relevance: All relevant items have to be modeled. Non-relevant items have to be left out, i.e. the value of the model doesn't decline if these items are removed.

3. Principle of Cost vs. Benefit: The amount of effort to gather the data and produce the model must be balanced against the expected benefit.

4. Principle of Clarity: The model must be understandable and usable. The required knowledge for understanding the model should be as low as possible.

5. Principle of Comparability: A common approach to modeling ensures future comparability of different models that have been created independently from each other.

6. Principle of Systematic Structure: Models produced in different views should be capable of integration. Interfaces need to be designed to ensure interoperability.

Here, an interesting point arises: Since typically modeler and model user are different entities, which exhibit different perspectives on the process, a good model for the modeler may be very different from a good model for the model user. For example, a detailed model may reflect reality very well, but it may be too complex to analyze it in simulations. Hence, modeling must be in line with the usage, and the quality of a model is determined by the degree it meets the requirements of the model user (“fitness for use”).

2.3 Simulation

Within this lecture, our intended use is simulation where we are going to evaluate a model over time. Utilizing time allows us to consider not only instances of inputs $\mathbf{u}$ but entire sequences $\mathbf{u}(\cdot)$ where the input is no longer a single value but a map $\mathbf{u} : \mathcal{T} \to \mathcal{U}$. To evaluate such sequences, we require the notion of a simulator to connect points over time:

Definition 2.10 (Simulator).
Consider a model Σ_M, a state $\mathbf{x}_0$ at time t_0 and a terminal time t_f to be given. Suppose that t_0 and t_f are the lower and upper bounds of a set $\mathbf{T} \subset \mathcal{T}$ and an input sequence $\mathbf{u}(t)$, $t \in \mathbf{T}$ to be given. We call a method $\Phi : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}$ a *simulator* if

$$\mathbf{x}(t) = \Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \tag{2.3}$$

holds for any $t \in \mathbf{T}$.

More plastically speaking, a simulator evaluates the model — and thereby depends on it — for given initial value $(t_0, \mathbf{x}_0)$, input sequence $\mathbf{u}(\cdot)$ and parameters p at any future time instant $t \in \mathbf{T}$ revealing the output of the model.

Remark 2.11
Here, we like to stress that a simulator is not unique, i.e. different methods can be applied. If the model is given by a set of differential equations, then both the analytic solution or numerical integrators or Koopman operators may be used as simulators.

Task 2.12
Consider the differential equation

$$\dot{\mathbf{x}}(t) = \mathbf{x}(t). \tag{2.4}$$

Define an analytical and a numerical simulator.

Solution to Task 2.12: The analytical solution of (2.4) is given by

$$\mathbf{x}(t) = C \cdot \exp(t)$$

which can be identified using initial values $\mathbf{x}(t_0) = \mathbf{x}_0$ to

$$\mathbf{x}(t) = \mathbf{x}_0 \cdot \exp(t - t_0).$$

Hence, we obtain

$$\Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot)) := \mathbf{x}_0 \cdot \exp(t - t_0).$$

A numerical solution of (2.4) can be derived using the Euler method

$$\mathbf{x}(t) = \mathbf{x}(t_0) + \dot{\mathbf{x}}(t_0) \cdot (t - t_0)$$

which again can be applied for given initial values $\mathbf{x}(t_0) = \mathbf{x}_0$. In this case, we obtain

$$\Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot)) := \mathbf{x}_0 + \dot{\mathbf{x}}(t_0) \cdot (t - t_0)\,.$$

Both are simulators but will reveal different results. For this reason, it is important to select the right solution method for the task at hand.

In contrast to simulators, simulation is makes use of not only a specific initial value, time interval and input sequence but allows to consider entire sets of the latter.

Definition 2.13 (Simulation).
Consider a model Σ_M and a simulator Φ to be given. Suppose sets of initial values $\mathcal{X}_0 \subset \mathcal{X}$, initial times $\mathcal{T}_0$ and terminal times $\mathcal{T}_f$ as well as input sequences sets $\mathcal{U}^{\mathbf{T}} := \{\mathbf{u}(t) \mid t \in \mathbf{T}\}$ and parameters $\mathcal{P}$ to be given. Then we call a map $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ given by

$$\Theta(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) = \Big\{\Phi(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \mid (t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p) \in \mathcal{T}_f \times \mathcal{T}_0 \times \mathcal{X}_0 \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}\Big\} \tag{2.5}$$

a *simulation*.

Based on the simulation results, an evaluation to assess the results is conducted using so called key performance indicators, which are defined on a high level via ISO 22400 [10]:

> A key performance criterion is a function, which measures defined information retrieved from the system/process or model against a standard.
>
> ISO 22400 (2014)

Similar to the high level definition of a system/process, in practice we require a more rigorous definition to actually calculate:

Definition 2.14 (Cost function).
We call a key performance criterion given by a function $\ell : \mathcal{X} \to \mathbb{R}_0^+$ a *cost function*.

Task 2.15
Design a cost function to compute the square deviation from a target $\mathbf{x}^{ref}$.

Solution to Task 2.15: Given the target value $\mathbf{x}^{\text{ref}}$, the standard squared deviation is given by

$$\ell(\mathbf{x}) := \left(\mathbf{x} - \mathbf{x}^{\text{ref}}\right)^2.$$

In that case the costs are minimal if the state of the model/system approaches the wanted target value $\mathbf{x}^{\text{ref}}$.

The value of a key performance criterion reveals a snapshot only, i.e. the evaluation at one time instant $t \in \mathcal{T}$. To obtain the performance, we need to evaluate it over the operating period of the system. Since by doing so we define a function of a function, this is referred to as a functional.

Definition 2.16 (Cost functional and assessment).
Consider a key performance criterion $\ell : \mathcal{X} \to \mathbb{R}_0^+$. Then we call

$$J(\Theta(\Sigma_M, \Phi)(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p)) := \int\limits_{t_0}^{t} \ell(\mathbf{x}(\tau)) d\tau \tag{2.6}$$

cost functional for the tupel $(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p)$. The combination of cost functionals for all tupels of the simulation $J(\Theta(\Sigma_M, \Phi))$ is called assessment.

Task 2.17
Define the mean of a simulation.

Solution to Task 2.17: An assessment using the mean over all simulations in the set $\mathcal{S} := \mathcal{T}_f \times \mathcal{T}_0 \times \mathcal{X}_0 \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ is given by

$$J(\Theta(\Sigma_M, \Phi) := \sum_{(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot)) \in \mathcal{S}} \frac{1}{\sharp \mathcal{S}} J(\Theta(\Sigma_M, \Phi)(t, t_0, \mathbf{x}_0, \mathbf{u}(\cdot), p))$$

which is an all-out simulation of all possible combinations in the set $\mathcal{S}$.

Remark 2.18
Note that similar to the mean, also other statistical parameters such as covariance, quantiles and statistical test can be performed.

Having defined the basic terms we require for simulation, we next introduce examples which highlight the different application scenarios outlined in Section 1.1.

2.4 Example

In the following, we will introduce the primary example, which we are going to use throughout the lecture. To this end, we consider the system of a quartercar test bench as depicted in Figure 2.4. This system is already complex but simple enough to understand its parameters and dynamics intuitively.
The system shall allow us to excite the test bench via the input $u(\cdot)$, which resembles road conditions. As the input sequence can be structured arbitrarily, optimization for untrained input is possible and in particular it is impossible to simulate all possible input sequences (as there exist infinitely many of them).
For the test bench, we assume that only vertical movements are considered. Moreover, we model the system such that the chassis is modeled as mass m_1 at position y_1, the suspension is modeled using a spring and a damper element s_1, d_1. Similarly, the wheel and the axis are modeled as mass m_2 at position y_2, the wheel is modeled using a spring and a damper element s_2. Last, road undulations are modeled via the road height function $u(t)$.
In order to apply d'Alemberts principle of equality of forces, we first model the individual forces

$$m_i \dot{v}_i^m(t) = F_i^m(t), \quad d_i v_i^d(t) = F_i^d(t), \quad s_i y_i^s(t) = F_i^s(t)$$

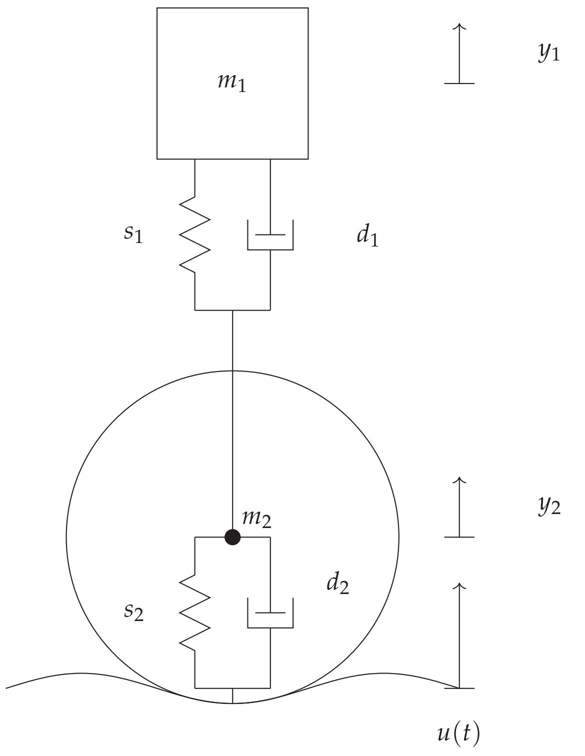


Figure 2.4: Schematic drawing of a quarter car test bench

for $i = 1, 2$ where we used

$$\begin{aligned}
v_1^m(t) &= \dot{y}_1(t), & v_1^d(t) &= v_1(t) - v_2(t), & v_1^s(t) &= y_1(t) - y_2(t), \\
v_2^m(t) &= \dot{y}_2(t), & y_2^d(t) &= v_2(t) - \dot{u}(t), & y_2^s(t) &= y_2(t) - u(t).
\end{aligned}$$

Now, we can combine the equations to describe the forces in all masses. To this end, the direction of the forces has to be treated carefully. In m_1, the force F_1^m points into the upwards direction: Since m_1 is the upper end of the attached spring and damper, F_1^d and F_1^s also point upwards. Hence, in m_1 we obtain

$$F_1^m + F_1^d + F_1^s = 0.$$

In m_2, forces F_1^d, F_1^s point downwards, all other forces upwards and we obtain

$$F_2^m - F_1^d - F_1^s + F_2^d + F_2^s = 0.$$

Combined, we have

$$\begin{aligned}
0 &= F_1^m + F_1^d + F_1^s \\
&= m_1 \dot{v}_1^m(t) + d_1 v_1^d(t) + s_1 y_1^s(t)
\end{aligned}$$

$$= m_1\ddot{y}_1(t) + d_1(\dot{y}_1(t) - \dot{y}_2(t)) + s_1(y_1(t) - y_2(t))$$

and

$$\begin{aligned}
0 &= F_2^m - F_1^d - F_1^s + F_2^d + F_2^s \\
&= m_2\dot{v}_2^m(t) - d_1 v_1^d(t) - s_1 y_1^s(t) + d_2 v_2^d(t) + s_2 y_2^s(t) \\
&= m_2\ddot{y}_2(t) - d_1(\dot{y}_1(t) - \dot{y}_2(t)) - s_1(y_1(t) - y_2(t)) + d_2(\dot{y}_2(t) - \dot{u}(t)) + s_2(y_2(t) - u(t)).
\end{aligned}$$

These equations display two second order differential equations and can be reformulated as a system of four first order differential equations.

Remark 2.19
Note that also a Lagrangian or Hamiltonian approach to derive the equations of motion can be used, yet the outcome will always be equivalent.

The above model is still rather simple and can be adapted using the parameters of the dampers and springs. So depending on the requirement of simulation detail, the results may be sufficiently accurate. Still, the model is complicated enough such that for a simulation one would choose a numerical solver.

In the following chapter, we will focus on description methods for such models and connect these to solution methods, which provide us with simulators.

CHAPTER 3 SOLUTION METHODS FOR DYNAMICAL SYSTEMS

In todays practice, dynamical systems are present in almost any scientific area. These systems arise from modeling of systems as outlined in the previous chapter, and may take several forms. Generally speaking, the (mathematical) description of models varies depending on the considered time, space and amplitude properties, cf. Figure 3.1 for a rough overview on these characteristics.

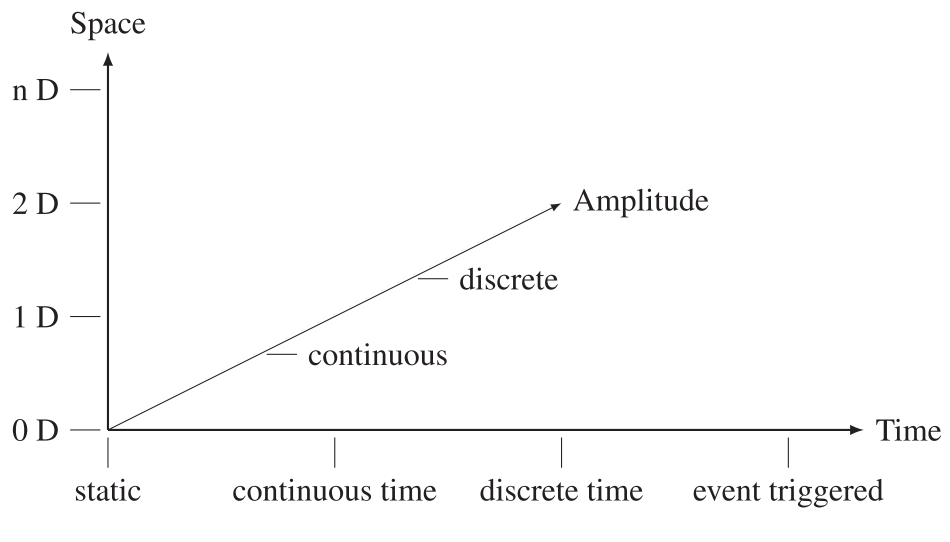


Figure 3.1: Dimensions of model characteristics

Within this lecture, we focus on models given by differential equations. These models arise naturally for mechatronic systems during modeling via, e.g. d'Alemberts principle or the Lagrangian or Hamiltonian approach. In the following Section 3.1, we recall important aspects from differential equations before stating numerical methods to solve the arising initial value problem in Sections 3.2–3.4. Last, we tackle the point of automated deployment of computations of solutions in Section 3.5. As such, we deal with the deepest point of the V model highlighted in Figure 3.2.

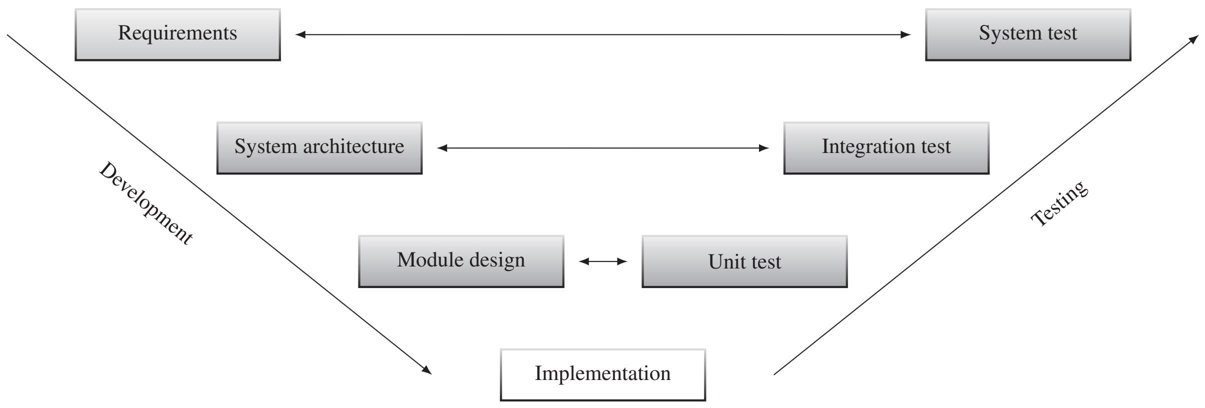


Figure 3.2: V model for system development

For an in-depth understanding, we refer to the books of Deuflhard [4] and Stoer [1].

3.1 Differential equations

As indicated by Figure 3.1, dynamical systems differ regarding structural parameters such as time, space and amplitude:

- Regarding time, we start off with static models, which are characterized by the fact that inputs, outputs and measurements of the system are available. In contrast to that, continuous time models exhibit data streams being received continuously. Discrete time models differ from that by the availability of data, which is received at certain, not necessarily equidistant time instances. Last, event triggered models require issues to trigger receiving data.

- Regarding space, models may vary from a simple connection to complex systems.

- Regarding amplitude, models may differ regarding continuous spaces like e.g. mass and discrete spaces such as gear shifts.

Within this section, we focus on models Σ_M given by differential equation (systems) of first order subject to continuous time, i.e. $\mathcal{T} = \mathbb{R}$, high dimensional space and real amplitude, i.e. $\mathcal{X} = \mathbb{R}^{n_x}$. Such an ordinary differential equation relates the derivative of a function $\mathbf{x} : \mathcal{T} \to \mathcal{X}$ with its one-dimensional argument and the function itself. More formally:

Definition 3.1 (Ordinary Differential Equation).
An *ordinary differential equation* in $\mathcal{X} = \mathbb{R}^{n_x}$, $n_x \in \mathbb{N}$, is given by the dynamic

$$\dot{\mathbf{x}}(t) := \frac{d}{dt}\mathbf{x}(t) = f(t, \mathbf{x}(t)) \tag{3.1}$$

where $f : \mathbb{D} \to \mathcal{X}$ is a continuous function with open subset of $\mathbb{D} \subset \mathcal{T} \times \mathcal{X} = \mathbb{R} \times \mathbb{R}^{n_x}$.

Task 3.2
Models derived via the Lagrangian approach typically form ordinary differential equation systems of second order

$$\ddot{\tilde{\mathbf{x}}}(t) = \tilde{f}(t, \tilde{\mathbf{x}}(t)). \tag{3.2}$$

Reform the latter in the form (3.1).

Solution to Task 3.2: System (3.2) can be reformulated equivalently to obtain a system of first order differential equation

$$\dot{\mathbf{x}}(t) = \begin{pmatrix} \dot{\mathbf{x}}_1(t) \\ \dot{\mathbf{x}}_2(t) \end{pmatrix} = \begin{pmatrix} \mathbf{x}_2 \\ \tilde{f}(t, \mathbf{x}_2(t)) \end{pmatrix} = f(t, \mathbf{x}(t))$$

where $\mathbf{x}_2 = \dot{\tilde{\mathbf{x}}}$.

The solution of (3.1) is a continuously differentiable function $\mathbf{x} : \mathcal{T} \to \mathcal{X}$, which satisfies (3.1). In general, we will use the following denomination throughout the script:

- The independent variable t is referred to as time, although other interpretations are possible.
- For the time derivative $\frac{d}{dt}\mathbf{x}(t)$ we use the abbreviation $\dot{\mathbf{x}}(t)$.
- The function $\mathbf{x}(t)$ is called *solution* at time t and the entirety $\mathbf{x}(\cdot)$ is called *trajectory*.
- If the function f is independent of t, i.e. $\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$, then the differential equation is called *autonomous*.

An ordinary differential equation typically possesses infinitely many solutions, cf. Task 2.12. To obtain a unique solution, we have to introduce a constraint, the so called initial value constraint. Combined with the differential equation (3.1), this reveals the so called initial value problem:

Definition 3.3 (Initial Value Problem).
Consider a function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ together with values $t_0 \in \mathcal{T}$ and $\mathbf{x}_0 \in \mathcal{X}$ to be given. Then the *initial value problem* is to find the solution satisfying the differential equation

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t)) \tag{3.1}$$

and the initial value condition

$$\mathbf{x}(t_0) = \mathbf{x}_0. \tag{3.3}$$

Here, the time $t_0 \in \mathcal{T}$ is called *initial time* and the value $\mathbf{x}_0 \in \mathcal{X}$ is called *initial value*. Both the pair $(t_0, \mathbf{x}_0)$ and equation (3.3) are called *initial condition*.

Remark 3.4
A continuously differentiable function $\mathbf{x} : \mathbb{D} \to \mathcal{X}$ *solves the initial value problem* (3.1), (3.3) *for some* $t_0 \in \mathcal{T}$ *and* $\mathbf{x}_0 \in \mathcal{X}$ *if and only if for each* $t \in \mathcal{T}$ *the integral equation*

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t_f} f(\tau, \mathbf{x}(\tau))\, d\tau \tag{3.4}$$

holds. This follows directly by integration of (3.1) *with respect to* t *or via differentiation of* (3.4) *with respect to* t *using the central theorem of differentiation and integration. Note that since continuity of* $\mathbf{x}(t)$ *on the right hand side of* (3.4) *implies continuous differentiability of the right hand side, each continuous function* $\mathbf{x}(t)$ *satisfying* (3.4) *is automatically continuously differentiable.*

Under certain conditions, existence and uniqueness of a solution to the problem from Definition 3.3 can be shown. This is the so called Lipschitz condition

Definition 3.5 (Lipschitz Condition).
Consider a function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. Then f is called *Lipschitz* in its second argument, if for each compact set $K \subset \mathcal{T} \times \mathcal{X}$ there exists a constant $L > 0$ and

$$\|f(t, x) - f(t, y)\| \leq L\|x - y\| \tag{3.5}$$

holds for all $t \in \mathcal{T}$ and all $x, y \in \mathcal{X}$ with $(t, x), (t, y) \in K$.

Task 3.6

Show that any function $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ is Lipschitz if it is continuously differentiable in its second argument.

Solution to Task 3.6: Since f is continuously differentiable, we have that there exists $K \in \mathbb{R}_0^+$ such that

$$\lim_{x \to y} \frac{\|f(t,x) - f(t,y)\|}{\|x - y\|} \leq K$$

holds. Now we can set $L := K$ and multiply both sides by $\|x - y\|$ showing the assertion.

Using this property, we can show the following:

Theorem 3.7 (Existence and Uniqueness).
Consider a differential equation (3.1) *with $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. Moreover, f is considered to be continuous and Lipschitz continuous in the second argument. Then for each initial condition $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$, there exists a unique solution $\mathbf{x}(t; t_0, \mathbf{x}_0)$ of the initial value problem* (3.1), (3.3). *This solution is defined for all t from an open maximal interval of existence $I_{t_0,\mathbf{x}_0}$ with $t_0 \in I_{t_0,\mathbf{x}_0}$.*

Note that by Task 3.6, the following holds:

Corollary 3.8 (Simplified Existence and Uniqueness).
Consider a differential equation (3.1) *with $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. If f is continuously differentiable in its second argument, then the assertion of Theorem 3.7 holds.*

Here, we like to note that the dynamic reveals a *flow* of the system at hand, whereas a *trajectory* is bound to a specific initial value and input sequence. The following Figure 3.3 illustrates the idea of flow and trajectory. In this case, the flow is colored to mark its intensity whereas the arrows point into its direction. The trajectory is evaluated for a specific initial value and „follows“ the flow accordingly.
Note that at the boundary of the interval of existence $I_{t_0,\mathbf{x}_0}$ the solution ceases to exist. If the interval is bounded, then there are two possible reasons for that: For one, the solution may diverge, or secondly the solution converges to a boundary point of $\mathcal{T} \times \mathcal{X}$. In the remainder of this script, we will always assume that the assumptions of Theorem 3.7 are met without explicitly stating it.

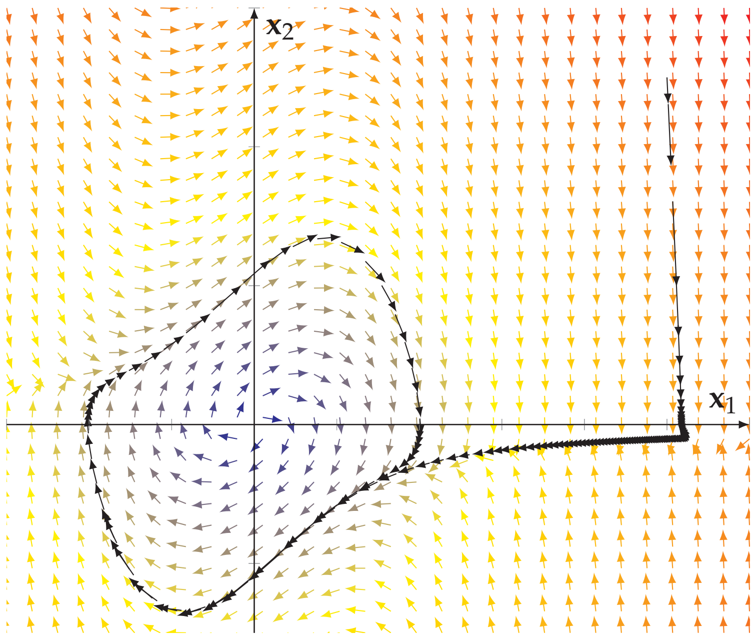


Figure 3.3: Sketch of a dynamic flow and a trajectory

Remark 3.9
1. A consequence of Theorem 3.7 is the so called cocycle property. This property states that for $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$ and two time instances $t_1, t \in \mathbb{R}$, we have

$$\mathbf{x}(t; t_0, \mathbf{x}_0) = \mathbf{x}(t; t_1, \mathbf{x}_1) \tag{3.6}$$

with $\mathbf{x}_1 = \mathbf{x}(t_1; t_0, \mathbf{x}_0)$ given that all terms are defined according to Theorem 3.7.
2. Another consequence is that two solutions cannot intersect, as they would have to coincide for all times.
3. Some ordinary differential equations can be solved analytically via various methods. In general, this is not true and numerical methods must be used for this purpose. Yet, one typically not only applies numerical methods, but tries to show certain properties of the solution analytically.

Table 3.1: Advantages and disadvantages of differential equations

Advantage	**Disadvantage**
✓ Allows to model dynamics	✗ May require solvers
✓ Provides unique solutions	✗ Requires identification

Based on ordinary differential equations and the initial value problem, we next introduce numerical methods to compute a solution of the initial value problem. These methods can later be used as simulators in the sense of Definition 2.10 in simulations.

Remark 3.10
We like to note that apart from numerical methods also analytical methods can be used. Amongst these, the Fourier method (also called separation of variables) is the most prominent one. Still other methods like direct integration, substitution or variation of parameters exist. These methods are outside the scope of this lecture. While being exact, these methods exhibit the downside of not being generally applicable.

In the following, we will concentrate on a particular class of numerical solution methods, which is already indicated in the class diagram sketched in Figure 3.4.

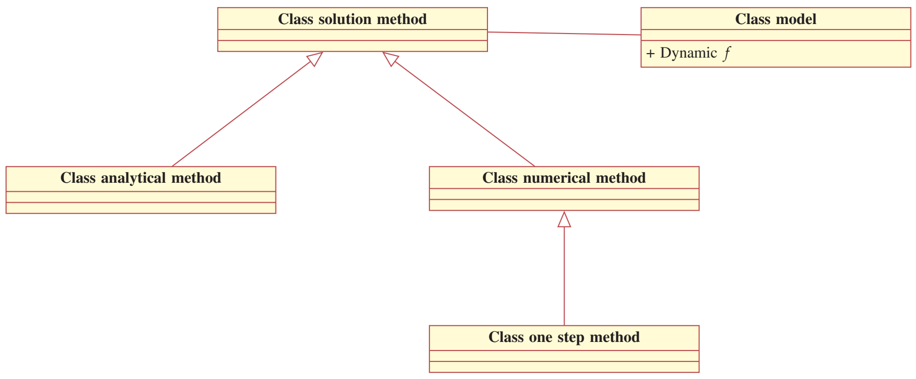


Figure 3.4: Sketch of UML diagram of class solution method

We like to stress that any solution method is independent from the model. While not all solution methods can be applied to specific models, they are designed to compute a solution for particular classes of models. The separation of solution method and model will be relevant in the upcoming Chapter 4 where we will focus on simulation and handling results.

3.2 Explicit methods

The most prominent and also most simple class of numerical methods are one step methods. The key characteristic of these methods is that they completely rely on the solution computed in the previous step. Multi step methods, on the other hand, consider a series of solutions for previous steps. Both classes have in common that they operate using fixed step sizes, that is sampled time instances. Utilizing our Definition 2.3 of time, we obtain:

Definition 3.11 (Sampled time).
Consider a time set $\mathcal{T} = \mathbb{R}$, a base time $t_0 \in \mathcal{T}$ and a number $T \in \mathbb{R}$. Then we call the set

$$\mathbb{T} := \{t_i \in \mathcal{T} \mid t_i = t_0 + i \cdot T,\ i \in \mathbb{Z}\} \tag{3.7}$$

sampled time set or *time grid*. Moreover, we call T *sampling period*.

As outlined, the idea of one step methods is to compute the solution of a differential equation based at a time instant using the solution at the previous one. In general, we define the following:

Definition 3.12 (One step method).
Suppose a map $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$ and a parameter $h \in \mathbb{R}$ to be given. Then we call

$$\mathbf{x}(t_{i+1}) = \Psi(h, \mathbf{x}(t_i)) \tag{3.8}$$

a *one step method* and h *step size*.

Technically, a one step method considers only one step from t_i to t_{i+1} and not the computation of a solution. Therefore, a numerical solution method is the generalization over time:

Definition 3.13 (Numerical solution method).
Consider a map $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$, initial conditions $(t_0, \mathbf{x}(t_0)) \in \mathbb{R} \times \mathcal{X}$ and a terminal time $\overline{T} \in \mathcal{T}, \overline{T} > t_0$ to be given. Then we call a map $\Psi : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ defining

$$\mathbf{x}(\overline{T}) = \Psi(\overline{T}, t_0, \mathbf{x}(t_0)) \tag{3.9}$$

a *numerical solution method*.

Remark 3.14
Note that in many cases step size and sampling period are identical, i.e. $h = T$.

Example 3.15
Given a differential equation (3.1)*, the most simple one step methods are the so called Euler method*

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + h \cdot f(t_i, \mathbf{x}(t_i)),$$

and the Heun method

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + \frac{h}{2} \cdot \left(f(t_i, \mathbf{x}(t_i)) + f\left(t_i + h, \mathbf{x}(t_i) + h \cdot f(t_i, \mathbf{x}(t_i))\right)\right).$$

To apply any numerical approximation method for differential equations, we must make sure that the approximation error stays bounded with respect to the chosen sampling period h. To this end, we want the approximation to converge to the true solution if the sampling period is reduced, i.e. $h \to 0$. To ensure convergence, two properties are required:

Definition 3.16 (Consistency).
Consider a one step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$. We call the method *consistent* if there exist $C, p \in \mathbb{R}_0^+$ such that

$$\|\Psi(h, \mathbf{x}_0) - \mathbf{x}(t_0 + h; t_0, \mathbf{x}_0)\| \leq C \cdot h^{p+1} \tag{3.10}$$

holds for all initial values $t_0 \in \mathcal{T}$, $\mathbf{x}_0 \in \mathcal{X}$ and all at least p times continuously differentiable models $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and all $0 < h \leq h^\star$. We call p the *order of consistency*.

Task 3.17
Assess whether consistency is a local or global property with respect to a trajectory.

Solution to Task 3.17: By definition, consistency means that the one step method is bounded in each single step. For this reason, it represents a local error and therefore a local property.

In the autonomous case, that is the dynamic f is independent from time, consistency can by checked using the following:

Lemma 3.18 (Consistency check).
Consider a one step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and suppose the initial value problem to be autonomous. Then the method is consistent if

$$\lim_{h \to 0} \left\| \frac{\Psi(h, \mathbf{x}) - \mathbf{x}}{h} - f(\mathbf{x}) \right\| = 0. \tag{3.11}$$

Task 3.19
Show that the Euler method is consistent.

Solution to Task 3.19: We directly obtain

$$\left\| \frac{\Psi(h,\mathbf{x}) - \mathbf{x}}{h} - f(\mathbf{x}) \right\| = \| f(\mathbf{x}) - f(\mathbf{x}) \| = 0.$$

Having dealt with the single step error, we have to include that any one step method by definition will use the possibly error prone value to continue calculating. To cope with this source, we define the following:

Definition 3.20 (Stability).
Suppose a one step method $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a constant $M > 0$ to be given. Then we call Ψ to be stable if

$$\|\Psi(h,\mathbf{x}_1) - \Psi(h,\mathbf{x}_2)\| \leq (1 + h \cdot M) \cdot \|\mathbf{x}_1 - \mathbf{x}_2\| \tag{3.12}$$

holds for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}$ and all $0 < h \leq h^\star$.

Task 3.21
Show that the Euler method is stable.

Solution to Task 3.21: Similar to consistency, we directly obtain

$$\begin{aligned}\|\Psi(h,\mathbf{x}_1) - \Psi(h,\mathbf{x}_2)\| &= \|\mathbf{x}_1 + h \cdot f(\cdot,\mathbf{x}_1) - \mathbf{x}_2 - h \cdot f(\cdot,\mathbf{x}_2)\| \\ &= \|\mathbf{x}_1 - \mathbf{x}_2\| + h \cdot \|f(\cdot,\mathbf{x}_1) - f(\cdot,\mathbf{x}_2)\| \leq (1 + h \cdot L) \cdot \|\mathbf{x}_1 - \mathbf{x}_2\|\end{aligned}$$

where L is the Lipschitz constant of f and we get $M = L$.

Now, these two properties can be combined to obtain convergence of the method:

Theorem 3.22 (Convergence).
Suppose a one step method $\Psi : \mathbb{R} \times \mathcal{X} \to \mathcal{X}$ of order p, which is consistent and stable and let $(t_0, \mathbf{x}_0)$ be given initial conditions. Then for each $\overline{h} \in \mathbb{R}$ there exists a constant $K(\overline{h}) > 0$ such that

$$\left\| \mathbf{x}(t_i) - \Psi^i(h, \mathbf{x}_0) \right\| \leq K(\overline{h}) \cdot h^p \tag{3.13}$$

holds for all $i \in \mathbb{N}$ with $i \cdot h \leq \overline{h}$ and all at least p times continuously differentiable models $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$.

Convergence can be seen as the global error of a one step method. What we can take from Theorem 3.22 together with Lemma 3.18 is that Lipschitz plus consistency (of order p) gives us convergence (of order $p-1$).

Table 3.2: Advantages and disadvantages of one step methods

Advantage	Disadvantage
✓ Provides universal method for ODEs	✗ May diverge quickly for large h
✓ Distinguishes local/global error	✗ Order diminishes by 1 along trajectory

The idea of the Heun method can be continued to derive methods of higher order. Similar as for the Heun method, these methods require additional evaluations of the dynamic f. These methods are combined in the class of so called Runge-Kutta methods:

Definition 3.23 (Explicit Runge-Kutta class methods).
Suppose a map $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a step size $h \in \mathbb{R}$ to be given. Let

$$\begin{aligned} k_1 &= f(t_j + c_1 \cdot h, \mathbf{x}(t_j)) && (3.14)\\ k_2 &= f(t_j + c_2 \cdot h, \mathbf{x}(t_j) + \alpha_{21} \cdot h \cdot k_1) && (3.15)\\ &\vdots && (3.16)\\ k_m &= f(t_j + c_m \cdot h, \mathbf{x}(t_j) + \alpha_{m1} \cdot h \cdot k_1 + \ldots + \alpha_{mm-1} \cdot h \cdot k_{m-1}). && (3.17) \end{aligned}$$

Then we call

$$\Psi(h, \mathbf{x}(t_j)) = \mathbf{x}(t_j) + h \cdot (\beta_1 \cdot k_1 + \ldots + \beta_m \cdot k_m) \tag{3.18}$$

an m-step *explicit Runge-Kutta* class method.

The Runge-Kutta class methods are defined uniquely by the coefficients α_{ij} and β_i.

Remark 3.24
In order to reveal a convergent (consistent and stable) behavior, the coefficients α_{ij} and β_i must satisfy certain conditions, which are outside the scope of this lecture.

Note that the number m refers to the required evaluations of the dynamics f and is in general not identical with the order of the method. In short, the methods are given by so called *Butcher tableaux*.

$$\begin{array}{c|ccccc}
c_1 & & & & & \\
c_2 & \alpha_{21} & & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_m & \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mm-1} & \\
\hline
 & \beta_1 & \beta_2 & \cdots & \beta_{m-1} & \beta_m
\end{array}$$

Table 3.3: Butcher tableaux of explicit Runge-Kutta class methods

Task 3.25
Assemble the Butcher tableaux of the Euler and Heun method given in Example 3.15.

Solution to Task 3.25: The Euler and Heun method are given by

$$\begin{array}{c|c}
0 & \\
\hline
 & 1
\end{array}
\qquad\qquad
\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}$$

Here, we like to point out that the classical Runge-Kutta method is a 4-step method is a special case of this class, cf. Table 3.4 for the respective Butcher tableaux.
From an algorithmic point of view, explicit one step methods are very simple to implement:

$$\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6}
\end{array}$$

Table 3.4: Butcher tableaux of the classical Runge-Kutta method

Algorithm 1 Explicit Runge-Kutta methods

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}_0^+$
Input: Terminal time $\overline{T}$
Input: Butcher tableaux of explicit $m$ step Runge-Kutta class method $\Psi$
 1: procedure CLASS EXPLICIT RUNGE-KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
 2:     $N \leftarrow (\overline{T} - t_0)/h$
 3:     for $j = 0, \ldots, N-1$ do
 4:         for $i = 1, \ldots, m$ do
 5:             $k_i = f(t_j + c_i \cdot h, \mathbf{x}(t_j) + \alpha_{i1} \cdot h \cdot k_1 + \ldots + \alpha_{ii-1} \cdot h \cdot k_{i-1})$
 6:         end for
 7:         $\mathbf{x}(t_{j+1}) \leftarrow \mathbf{x}(t_j) + h \cdot (\beta_1 \cdot k_1 + \ldots + \beta_m \cdot k_m)$
 8:         $t_{j+1} = t_j + h$
 9:     end for
10: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

While the method is simple in its implementation, the derivation of the coefficients is a very difficult task. We like to note that the order of convergence (and consistency) comes at a price, i.e. the higher the order the higher the required number of function evaluations, cf. Table 3.5.

Table 3.5: Order of consistence vs. number of function evaluations for one step methods

Order of consistence p	1	2	3	4	5	6	7	8	≥ 9
Minimal steps m	1	2	3	4	6	7	9	11	$\geq p+3$

In general, the following holds:

Theorem 3.26 (Bounded order of consistency for explicit methods).
Consider an explicit Runge-Kutta method with m steps. Then the order of consistency p of this method is bound by $p \leq m$.

Remark 3.27
At this point, we need to be careful: On the one hand, Theorem 3.26 assures that a higher order of consistency and thereby accuracy of the solution can be achieved. The expense we have to pay comes in number of steps m causing the dynamic f to be evaluated respectively often. Up till now, there exists no answer to the question which solver should be chosen for a specific problem.

On the implementation side, we know that the explicit Runge-Kutta method is a derivative of a general one step method. As such, it should be a derivative of a respective object. In the following Figure 3.5, we sketch the connection of both. Here, we already separated the class model containing the dynamics of our system.

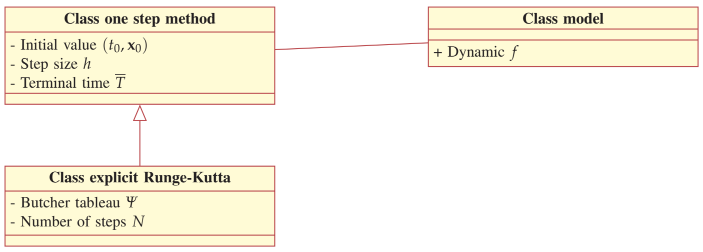


Figure 3.5: Sketch of UML diagram of class explicit Runge-Kutta

Remark 3.28
Note that Figure 3.5 is not complete and only a sketch of a possible class diagram.

Regarding explicit one step methods in general, we see the advantages and disadvantages outlined in Table 3.6.

Table 3.6: Advantages and disadvantages of explicit Runge-Kutta class methods

Advantage	Disadvantage
✓ Allows for simple implementation	✗ Requires multiple evaluations
✓ Utilizes intermediate steps	✗ Propagates intermediate error

3.3 Implicit methods

In contrast to explicit methods, for implicit methods a full stack of coefficients is used. The idea of using additional coefficients is to raise the bound on consistency of the method, that is to allow for a more accurate approximation with less intermediate function evaluations.
More formally, we define the following:

Definition 3.29 (Implicit Runge-Kutta class methods).
Suppose a map $\Psi : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ and a step size $h \in \mathbb{R}$ to be given. Let

$$k_i = f\left(t_i + c_i \cdot h, \mathbf{x}(t_i) + \sum_{j=1}^{m} \alpha_{ij} k_j\right) \tag{3.19}$$

define intermediate steps for $i = 1, \ldots, m$. Then we call

$$\Psi(h, \mathbf{x}(t_i)) = \mathbf{x}(t_i) + h \cdot \sum_{i=1}^{m} \beta_i \cdot k_i \tag{3.20}$$

an m-step *implicit Runge-Kutta* class method.

The method is called implicit as the values of k_i, $i = 1, \ldots, m$ is not longer a definition but requires the solution of a $m \cdot n_x$ dimensional nonlinear equation system. Similar to the explicit case, the Butcher tableau can be defined as in Table 3.7.

$$\begin{array}{c|cccc}
c_1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1m} \\
c_2 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2m} \\
\vdots & \vdots & \vdots & \ddots & \\
c_m & \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mm} \\
\hline
 & \beta_1 & \beta_2 & \cdots & \beta_m
\end{array}$$

Table 3.7: Butcher tableaux of implicit Runge-Kutta class methods

For these methods, we can show the following:

Theorem 3.30 (Bounded order of consistency for implicit methods).
Consider an implicit Runge-Kutta class method with m steps. Then the order of consistency p of this method is bound by $p \leq 2 \cdot m$.

Again similar to the explicit case, the implementation of implicit methods is a straight forward procedure as displayed in Algorithm 2.

Algorithm 2 Implicit Runge-Kutta methods

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}_0^+$
Input: Terminal time $\overline{T}$
Input: Butcher tableaux of implicit $m$ step Runge-Kutta class method $\Psi$
 1: procedure CLASS IMPLICIT RUNGE-KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
 2:     $N \leftarrow (\overline{T} - t_0)/h$
 3:     for $j = 0, \ldots, N-1$ do
 4:         $k \leftarrow$ SOLUTION NONLINEAR EQUATION SYSTEM($f, t_j, \mathbf{x}(t_j), h, \Psi, \varepsilon$)
 5:         $\mathbf{x}(t_j) \leftarrow \mathbf{x}(t_j) + h \cdot \sum_{i=1}^{m} \beta_i \cdot k_i$
 6:         $t_{j+1} = t_j + h$
 7:     end for
 8: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

While being a significant improvement regarding consistence/convergence, we can see from Algorithm 2 that implicit methods require the solution of the nonlinear equation system for the auxiliary variables k_i. The latter can be solved using the Banach fixpoint iteration. The latter, however, requires that the map is a contraction. Here, we use the abbreviations

$$k := \begin{pmatrix} k_1 \\ \vdots \\ k_m \end{pmatrix} \qquad \text{and} \qquad F(k) := \begin{pmatrix} f\left(t_j + c_1 \cdot h, \mathbf{x}(t_j) + \sum_{j=1}^{m} \alpha_{1j} k_j\right) \\ \vdots \\ f\left(t_j + c_m \cdot h, \mathbf{x}(t_j) + \sum_{j=1}^{m} \alpha_{mj} k_j\right) \end{pmatrix}.$$

In order to be a contraction, we require

$$\|F(k^j) - F(k^i)\| \leq K \|k^j - k^i\|$$

to hold. Here, we have the following:

Theorem 3.31 (Contraction of implicit Runge-Kutta methods).
Consider an implicit Runge-Kutta class method and suppose the dynamics f to be Lipschitz with

Lipschitz constant L. Then the constant K in

$$\|F(k^j) - F(k^i)\| \leq K\|k^j - k^i\| \tag{3.21}$$

can be given by $K = h \cdot L$. *If the step size is chosen such that* $K = h \cdot L < 1$, *then inequality* (3.31) *is a contraction.*

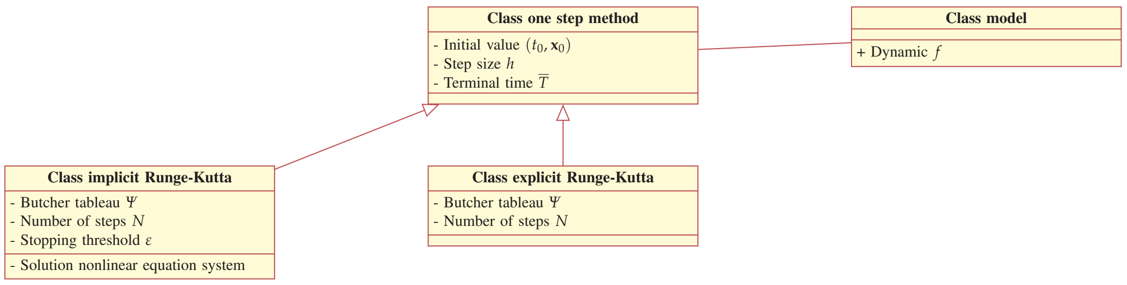
Given that the assumptions made in Theorem 3.31 hold, the following Algorithm 3 can be applied to solve the nonlinear equation system (3.19)

Algorithm 3 Full step iteration

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}_0^+$
Input: Butcher tableaux of implicit $m$ step Runge-Kutta class method $\Psi$
Input: Stopping threshold $\varepsilon \in \mathbb{R}^+$
1: procedure SOLUTION NONLINEAR EQUATION SYSTEM($f, t_0, \mathbf{x}_0, h, \Psi, \varepsilon$)
2:     $i = 1$ and $k^{i-1} \leftarrow 1, k^i \leftarrow 0$
3:     while $\|k^i - k^{i-1}\| \geq \varepsilon$ do
4:         $k^{i+1} = \begin{pmatrix} k_1^{i+1} \\ \vdots \\ k_m^{i+1} \end{pmatrix} \leftarrow \begin{pmatrix} f\left(t_0 + c_1 \cdot h, \mathbf{x}_0 + \sum_{j=1}^{m} \alpha_{1j} k_j^{i+1}\right) \\ \vdots \\ f\left(t_0 + c_m \cdot h, \mathbf{x}_0 + \sum_{j=1}^{m} \alpha_{mj} k_j^{i+1}\right) \end{pmatrix} = F(k^i)$
5:         $i \leftarrow i + 1$
6:     end while
7: end procedure
Output: Approximation $k$ of solution of nonlinear equation system
```

Similar to the explicit case, on the implementation side we can inherit similar items from the class one step method. Internally, the implicit methods require the stopping threshold and a function to solve nonlinear equation systems as in Algorithm 3. Figure 3.6 provides a respective sketch of the class diagram.

Here, we like to note that the order of consistence of the implicit Runge-Kutta class methods depends on the order of approximation of the solution of the nonlinear equation system. In particular, we require the local error bound (3.16) to hold, which renders ε to be dependent on h. Combined, we restate facts for implicit Runge-Kutta class methods outlined in Table 3.8.



Class one step method
- Initial value $(t_0, \mathbf{x}_0)$
- Step size h
- Terminal time $\overline{T}$
Class model
+ Dynamic f
Class implicit Runge-Kutta
- Butcher tableau Ψ
- Number of steps N
- Stopping threshold ε
- Solution nonlinear equation system
Class explicit Runge-Kutta
- Butcher tableau Ψ
- Number of steps N

Figure 3.6: Sketch of UML diagram of class implicit Runge-Kutta

Table 3.8: Advantages and disadvantages of implicit Runge-Kutta class methods

Advantage	Disadvantage
✓ Allows higher order of convergence	✗ Consistency depends on chosen error
✓ Reduces number of evaluations per order of consistency	✗ Requires solution of nonlinear equation system
	✗ Limits step size

3.4 Adaptive step size methods

For one step methods, our basic assumption was that the step size h is constant. As solutions to differential equations may exhibit regions with very fast and complex behavior as well as regions where this is not the case, constant step size is not efficient. In this case, efficient means that the number of required evaluations of the dynamic is as low as possible. For constant step size, always the worst case of fast/complex behavior must be assumed to guarantee a sufficiently good approximation.

The concept of adaptive stept size methods is to eliminate or at least reduce this downside. To this end, first one iteration with step size h is executed and the numerical error of the solution is estimated. Secondly, if the error is larger than a predefined threshold, then a smaller step size is calculated based on the estimated error and the iteration step is repeated. Lastly, if the error is smaller than the threshold, the step size is calculated based on the estimated error and applied for the next iteration step.

In order to compute an estimate of the error and a suitable new step size, the idea is to combine two one step methods of different order of consistency. Then we know that by construction the method of higher order is more reliant and can use both methods to correct one another. Since evaluating two different methods results in many evaluations of the dynamic, it is efficient to use methods for which function evaluations can be reuses – so called embedded methods.

Definition 3.32 (Embedded one step method).
Suppose two convergent one step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. Furthermore let the intermediate steps

$$k_i = f\left(t_i + c_i \cdot h, \mathbf{x}(t_i) + \sum_{j=1}^{m} \alpha_{ij} k_j\right) \tag{3.22}$$

be identical for both methods for $i = 1, \ldots, m$. Then the combination Ψ^1, Ψ^2 is called an embedded one step method.

Remark 3.33
By (3.22)*, the values α_{ij} and c_i are identical for both methods. Since the methods are of different order, we obtain that β_j are different for these methods.*

Two commonly found embedded methods are the Runge-Kutta $4(3)$ and the Dormand–Prince $5(4)$ method (in MATLAB called ode45) shown in Tables 3.9 and 3.10.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
1	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	
Ψ^1	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	0
Ψ^2	$\frac{1}{6}$	$\frac{2}{6}$	0	$\frac{1}{6}$

Table 3.9: Butcher tableaux of the Runge-Kutta $4(3)$ method

Before combining the latter to an algorithm, we first discuss and analyze the three described steps.

Error estimation

Since the step size can only be adapted for the current and for future iterations, it does not make sense to elaborate on errors, which already occurred in past iterations. Hence, we can focus on stability, i.e. one single step, and ignore convergence, i.e. the long run which is already given by construction of both one step methods.

$$\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
\Psi^1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
\Psi^2 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}$$

Table 3.10: Butcher tableaux of the Dormand-Prince $5(4)$ method

As we use two different one step methods of different order, we want to estimate the error of the lower order method. To this end, we compare the true solution to the solution of the lower order method Ψ^1

$$\varepsilon^1 := \mathbf{x}(t_{j+1}) - \Psi^1(h, \mathbf{x}(t_j))$$

where h is the current step size. Similarly, we obtain the error of the solution of the higher order method Ψ^2

$$\varepsilon^2 := \mathbf{x}(t_{j+1}) - \Psi^2(h, \mathbf{x}(t_j))$$

and the difference of both

$$\varepsilon := \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j)).$$

As ε is the only expression we can actually compute, it should be the basis for computing a step size adaptation. Now, we can utilize that Ψ^2 is of higher order. Hence, we have that

$$\lim_{h \to 0} \frac{\|\varepsilon^2\|}{\|\varepsilon^1\|} = 0.$$

Defining $\theta := \|\varepsilon^2\| / \|\varepsilon^1\|$, we can reformulate the latter fraction to

$$\frac{\|\varepsilon^2\|}{\|\varepsilon^1\|} = \frac{\|\varepsilon^1 - \varepsilon\|}{\|\varepsilon^1\|} = \theta \quad \Longleftrightarrow \quad \|\varepsilon^1 - \varepsilon\| = \|\varepsilon^1\| \cdot \theta.$$

Applying the triangle inequalities

$$\|\varepsilon^1\| - \|\varepsilon\| \leq \|\varepsilon^1 - \varepsilon\| \leq \|\varepsilon^1\| + \|\varepsilon\|$$

we obtain

$$\frac{1}{1+\theta}\|\varepsilon\| \leq \|\varepsilon^1\| \leq \frac{1}{1-\theta}\|\varepsilon\|.$$

Hence, for $h \to 0$ we have $\theta \to 0$ and the approximation

$$\|\varepsilon\| \approx \|\varepsilon^1\|,$$

holds. Therefore, we can use $\|\varepsilon\|$ as approximation of the true error $\|\varepsilon^1\|$ of the lower order method. This reveals

Theorem 3.34 (Approximated error for adaptive step size methods).
Consider two convergent one step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. If the step size h is sufficiently small, then the error of the lower order method can be approximated via the difference of the local errors of Ψ^1, Ψ^2

$$\varepsilon^1 \approx \varepsilon := \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j)) \tag{3.23}$$

for each iteration instant $t_j = t_0 + j \cdot h$.

Computation of step size

Based on the approximated error ε, an adaptation of the step size shall be computed. To this end, a threshold $\overline{\varepsilon}$ for the error needs to be predefined for the adaptive step size method.
From consistency, we know that the lower order method satisfies

$$\|\varepsilon^1\| \leq C \cdot h^{p+1}$$

where p is the known order of the method and C is unknown. Hence, in the worst case, the latter holds with equality. Now we can use our approximation $\varepsilon \approx \varepsilon^1$ and the worst case to obtain

$$\|\varepsilon\| \approx \|\varepsilon^1\| = C \cdot h^{p+1}$$

and can identify

$$C \approx \frac{\|\varepsilon\|}{h^{p+1}}.$$

The new step size h_{new} shall satisfy

$$C \cdot h_{\text{new}}^{p+1} \leq \overline{\varepsilon}$$

where we can use the identified constant C to obtain

$$C \cdot h_{\text{new}}^{p+1} \approx \frac{\|\varepsilon\|}{h^{p+1}} \cdot h_{\text{new}}^{p+1} \leq \overline{\varepsilon} \quad \Longleftrightarrow \quad h_{\text{new}} \approx \sqrt[p+1]{\frac{\overline{\varepsilon}}{\|\varepsilon\|}} \cdot h.$$

As this is still an approximation, in practice an additional safety constant $\eta = 0.9$ is used to derive the new step size. Combined, we obtain the following:

Theorem 3.35 (Adaptive step size computation).
Consider two convergent one step methods $\Psi^1, \Psi^2 : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$ of different order. Let p be the lower order of both methods and suppose the step size $h > 0$ to be sufficiently small for Theorem 3.34 to hold. To satisfy a given error threshold $\overline{\varepsilon} > 0$, the step size can be adapted using

$$h_{new} = \eta \cdot \sqrt[p+1]{\frac{\overline{\varepsilon}}{\|\varepsilon\|}} \cdot h. \tag{3.24}$$

with safety constant $\eta \in (0, 1)$.

Note that the threshold is guaranteed for the lower order method. Still, the solution of the higher order method is already available and most likely better in comparison. For this reason, typically the solution of the higher order method is used to define the iteration step in practice.

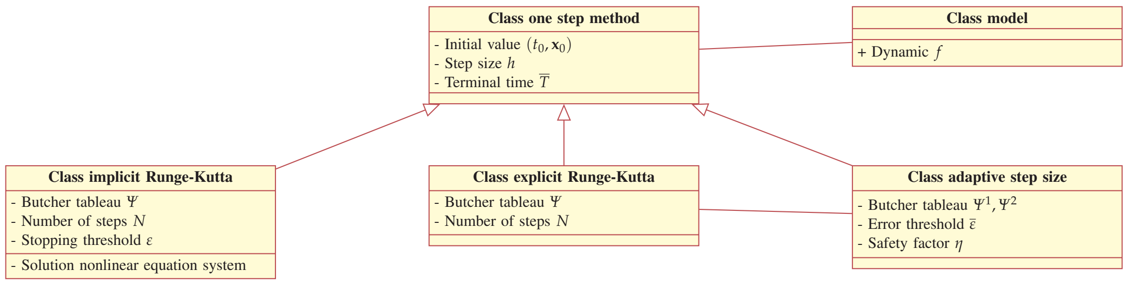
Remark 3.36
Regarding the next iteration step, an identical computation as in Theorem 3.35 can be used. In contrast to the current iteration step, however, not only a reduction of the step size but also an increase may occur.

Integrating all of the above, we obtain Algorithm 4.

Algorithm 4 Adaptive step size method

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Butcher tableaux of embedded method $\Psi^1, \Psi^2$
Input: Error threshold $\overline{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
 1: procedure CLASS ADAPTIVE STEP SIZE($f, t_0, \mathbf{x}_0, \overline{T}, \Psi^1, \Psi^2, \overline{\varepsilon}, \eta$)
 2:     $h \leftarrow \overline{T} - t_0$
 3:     while $t_j \neq \overline{T}$ do
 4:         if $t_j + h > \overline{T}$ then
 5:             $h \leftarrow \overline{T} - t_j$
 6:         end if
 7:         $t_{j+1} \leftarrow t_j + h$
 8:         $\Psi^1(h, \mathbf{x}(t_j)) \leftarrow$ CLASS EXPLICIT RUNGE-KUTTA($f, t_j, \mathbf{x}(t_j), h, t_{j+1}, \Psi^1$)
 9:         $\Psi^2(h, \mathbf{x}(t_j)) \leftarrow$ CLASS EXPLICIT RUNGE-KUTTA($f, t_j, \mathbf{x}(t_j), h, t_{j+1}, \Psi^2$)
10:         $\varepsilon \leftarrow \Psi^2(h, \mathbf{x}(t_j)) - \Psi^1(h, \mathbf{x}(t_j))$
11:         $h_{\text{new}} \leftarrow \eta \cdot \sqrt[p+1]{\frac{\overline{\varepsilon}}{\|\varepsilon\|}} \cdot h$
12:         if $\varepsilon > \overline{\varepsilon}$ then
13:             $h \leftarrow h_{\text{new}}$
14:         else
15:             $\mathbf{x}(t_{j+1}) := \Psi^2(h, \mathbf{x}(t_j))$
16:             $h \leftarrow h_{\text{new}}$
17:             $j \geq j + 1$
18:         end if
19:     end while
20: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

Utilizing Algorithm 4, we can update our schematic sketch of the class one step method as outlined in Figure 3.7. We like to note that it is not necessary to re-implement the explicit Runge-Kutta in order to be used in the adaptive step size method.



Class one step method
- Initial value $(t_0, \mathbf{x}_0)$
- Step size h
- Terminal time $\overline{T}$
Class model
+ Dynamic f
Class implicit Runge-Kutta
- Butcher tableau Ψ
- Number of steps N
- Stopping threshold ε
- Solution nonlinear equation system
Class explicit Runge-Kutta
- Butcher tableau Ψ
- Number of steps N
Class adaptive step size
- Butcher tableau Ψ^1, Ψ^2
- Error threshold $\overline{\varepsilon}$
- Safety factor η

Figure 3.7: Sktech of UML diagram of class one step method

Regarding adaptive step size methods, Table 3.11 summarizes advantages and disadvantages of such methods.

Table 3.11: Advantages and disadvantages of adaptive step size methods

Advantage	**Disadvantage**
✓ Adapts step size to dynamics	✗ Consistency depends on lower order
✓ Reduces number of evaluations	✗ Requires two one step methods
✓ Reuses intermediate results	✗ Requires embedding of methods

We like to note that the classes discussed so far in Algorithms 1, 2 and 4 can be combined in the class of one step methods. This is exactly, what we already did using the class diagrams in Figure 3.7. Algorithm 5 formalizes the latter.

Algorithm 5 One step methods

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}^+$
Input: One step method $\Psi$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\overline{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
 1: procedure CLASS ONE STEP METHOD($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
 2:     if $\Psi$ is explicit Runge-Kutta method then
 3:         $\mathbf{x}(\overline{T}) \leftarrow$ CLASS EXPLICIT RUNGE-KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
 4:     else if $\Psi$ is implicit Runge-Kutta method then
 5:         $\mathbf{x}(\overline{T}) \leftarrow$ CLASS IMPLICIT RUNGE-KUTTA($f, t_0, \mathbf{x}_0, h, \overline{T}, \Psi$)
 6:     else if $\Psi$ is adaptive step size method then
 7:         $\mathbf{x}(\overline{T}) \leftarrow$ CLASS ADAPTIVE STEP SIZE($f, t_0, \mathbf{x}_0, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
 8:     else if ... then
 9:     end if
10: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

We can already observe from the algorithms above that the interface for the adaptive step size method slightly differs from the explicit and implicit methods. Within an implementation, the respective function of the derived objects must therefore be modified.

3.5 Adaptation for deployment

As we have seen in the previous sections, one step methods are a suitable approach to numerically solve differential equations. Yet right at the beginning, we distinguished between the sampling time T of the system/process and the step size h of the method. If there are no requirements posed to sampling time and step size, the standard choice is to set $h = T$ and define the sampling time. Such an approach, however, can only be applied to pure software cases.

Remark 3.37
Upon separating step size and sampling time, there are three generic cases:

- *Unified grid: All systems are evaluated using a superset of grids. In simulation, this is a common approach despite inducing high computational load. In practice, however, measurement values are missing and the approach cannot be utilized.*
- *Splitting grids: Sampling and step size are considered separately and intermediate points are neglected on a vice versa basis. Again, this is possible in simulations. The computational load is greatly reduced compared to a unified grid, yet now at minimum two programs are required, which additionally need to be synchronized using, e.g., thread blockers. Similar due to unavailability of measurements, this is not realizable in practice.*
- *Debouncing: Here, zero order hold is used for measurements, i.e. held constant until a new measurement is available. Using this idea, the grids can be split but additionally a realization in practical applications is possible.*

In case there are restrictions for sampling time T, then a so called multiscale time grid can be used for deriving the solution of the differential equation (3.1).

Definition 3.38 (Multiscale time grid).
Consider sampling time $T \in \mathbb{R}_{>0}$ to be given defining a time grid $\mathbb{T}$ according to (3.7). Then for each $i \in \mathbb{N}$ the step size $h := T/i$ defines a multiscale time grid $\mathbb{T}_h$ satisfying $\mathbb{T} \subset \mathbb{T}_h$.

In practice, this is particularly useful. Applying a multiscale time grid allows us to apply numerical solution methods to systems containing components operating at a much slower speed T than the numerical methods. Upon implementation, we can utilize Algorithm 6 accordingly.
Considering the class diagrams we discussed so far, we see that the class multiscale defines the values of step size h, sampling time T, terminal time $\overline{T}$ and the initial values $(t_0, \mathbf{x}_0)$ to be used in the one step method. Hence, these values are shifted in the sketch given in Figure 3.8.

Algorithm 6 Multiscale application of one step methods

```
Input: Dynamic $f : \mathcal{T} \times \mathcal{X} \to \mathcal{X}$
Input: Initial value $(t_0, \mathbf{x}_0) \in \mathcal{T} \times \mathcal{X}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One step method $\Psi$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\overline{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
1: procedure CLASS MULTISCALE($f, t_0, \mathbf{x}_0, h, T, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
2:     $N_T \leftarrow (\overline{T} - t_0)/T$
3:     for $i = 0, \ldots, N_T$ do
4:         $\mathbf{x}(t_{i+1}) \leftarrow$ CLASS ONE STEP METHOD($f, t_i, \mathbf{x}(t_i), h, t_i + T, \Psi, \overline{\varepsilon}, \eta$)
5:     end for
6: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

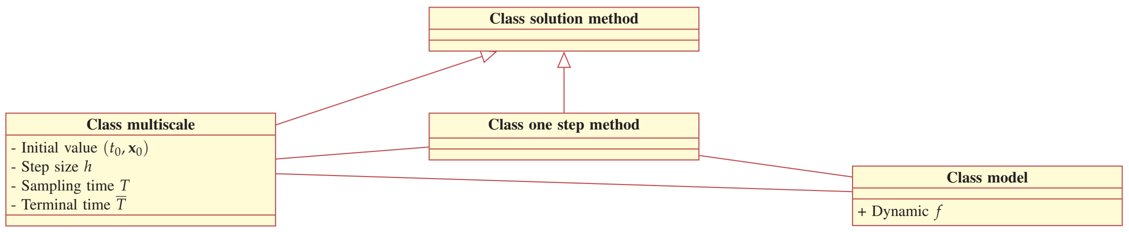


Figure 3.8: Sketch of UML diagram of class multiscale method

The multiscale application additionally allows us to use the cocycle property of differential equations, cf. Remark 3.9, and switch solutions methods at runtime.

Table 3.12: Advantages and disadvantages of multiscale methods

Advantage	Disadvantage
✓ Decouples step size and sampling	✗ Requires additional grid
✓ Utilizes cocycle property	✗ May require asynchronous runs
✓ Integrates previous methods	

Remark 3.39
Here, we only consider the case where the sampling time T is fixed. In practice, also the case of fixed sampling time T and bounded step size h exists. As there may not be a common multiscale structure, sampling and solution may be required to run asynchronously. Respective solutions are beyond the scope of this lecture.

CHAPTER 4

SIMULATION

In the previous Chapter 3 we discussed methods to compute solutions of differential equations. Within the present chapter, we will utilize these methods to analyze a given model via its solutions. In that case, we talk about simulation. The difference between model and simulator is given by its usage: A model is a representation of system whereas a simulator uses a model to generate its behavior. Hence, we are moving up the left side of the V model as indicated in Figure 4.1.

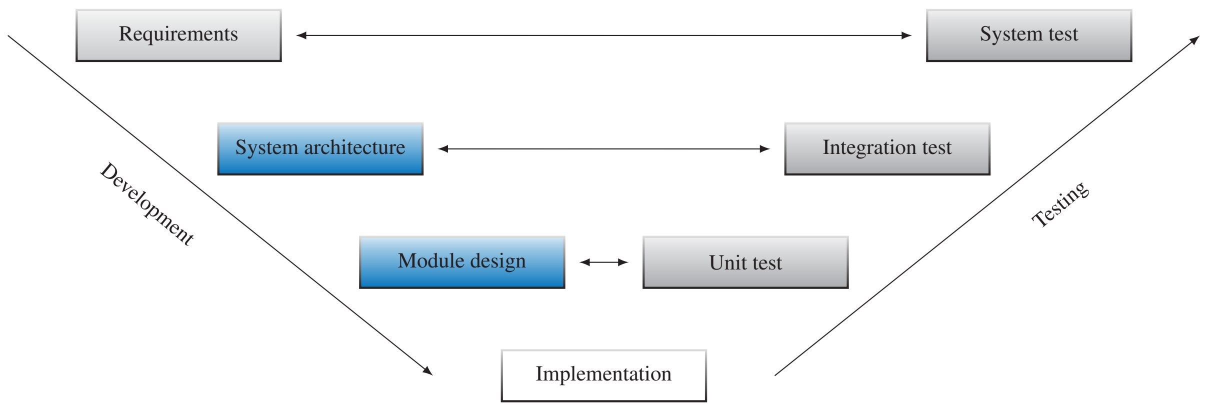


Figure 4.1: V model for system development

As we have seen in the previous chapter, we can apply one step methods to models, which satisfy the requirements of these methods. This provides us with two different possibilities to generate the behavior: For one, we can modify the parameters set by the solution method, and secondly we can consider parameters set within the model. In both cases, only those parameters may be changed, which do not compromise the properties of solution method and model, i.e. convergence (or order of convergence) for the method or order of the differential equation system for the model. Within this chapter, we first address the simulation problem itself and then discuss how a parametrization impacts on the structure of that problem in Section 4.1. Thereafter, we focus on processing results from a statistical and visualization point of view in Section 4.2. This leads us to identifying relations between parameters, which we discuss using sensitivities in Section 4.3. Concluding the chapter, we shortly discuss automation of these results.

4.1 Parametrization and Processing

In Chapter 3, the methods we derived aimed to solve the initial value problem from Definition 3.3. As we outlined before, we want to generate the behavior of the model for a set of parameters within both the model itself as well as the solution method.
Starting with the model itself, we need to make explicit which parameters we aim to modify. To the end, we introduce the parametrized control system:

Definition 4.1 (Parametrized control system).
We call a map $f : \mathcal{T} \times \mathcal{X} \times \mathcal{U} \times \mathcal{P} \to \mathcal{X}$ given by

$$\dot{\mathbf{x}}(t) = f(t, \mathbf{x}(t), \mathbf{u}(t), p) \tag{4.1}$$

a *parametrized control system.* We also refer to (4.1) as parametrized model Σ_M.

Within this definition, we distinguish between time dependent parameters, here $\mathbf{u}(\cdot)$ being a function of time, and time independent (constant) parameters indicated by p. The first denotation $\mathbf{u}(\cdot)$ typically refers to inputs of the system/model, which may be set from the outside to obtain properties such as, e.g., stability of the system. Within this lecture, we only refer to it as time dependent parameters.
As we can directly observe, the parametrized control system allows us to modify the dynamics of the model via our simulator given in Definition 2.10.

Remark 4.2
Note that a model as given in Definition 2.7 is more accurately a map from input to output, whereas the parametrized model (4.1) *maps to states. For completeness, one may add*

$$\mathbf{y}(t) = h(t, \mathbf{x}(t), \mathbf{u}(t), p)$$

to render Σ_M to be a map from input to output.

Recalling the definitions of a simulator and simulation, the simulator may additionally parametrized using the solution method as well as step size and sampling time. Based on that, we can now merge these with the solution methods and the parametrized model we defined, cf. Figure 4.2.

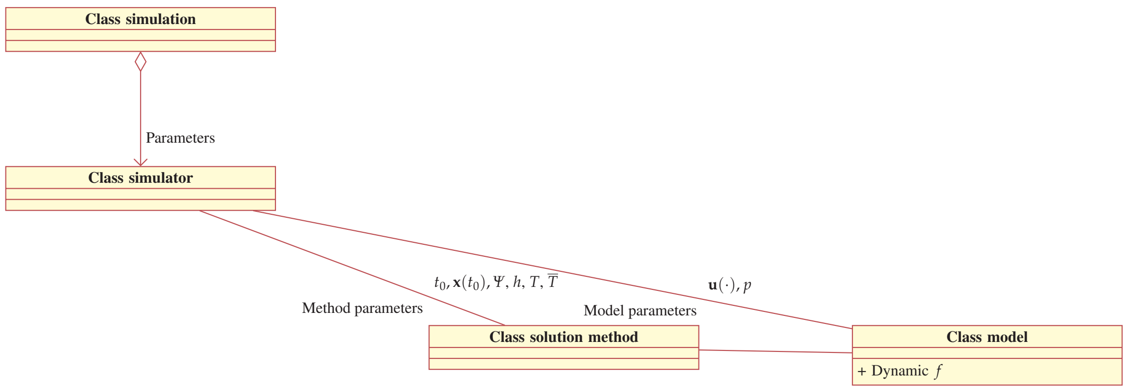


Figure 4.2: Sketch of UML diagram of class simulator

We like to stress that the connection drawn from class simulation to class simulator is an aggregation, i.e. an instance of class simulation may hold a variety of instances of class simulator. A practical implication of the latter is that a simulation may run several simulators at the same time, e.g. for different parameter combinations. Hence, an algorithmic implementation is rather straight forward as shown in Algorithms 7 and 8.

Algorithm 7 Prototype of simulator

```
Input: Model $\Sigma_M$
Input: Parameters $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One step method $\Psi$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\overline{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
1: procedure CLASS SIMULATOR($\Sigma_M, t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p, h, T, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
2:     $\Sigma_M \leftarrow$ CLASS MODEL($\mathbf{u}(\cdot), p$)
3:     $\mathbf{x}(\overline{T}) \leftarrow$ CLASS SOLUTION METHOD($\Sigma_M, t_0, \mathbf{x}(t_0), h, T, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
4: end procedure
Output: Endpoint of trajectory $\mathbf{x}(\overline{T})$
```

The core component we use to implement a simulation is a computer program.

Definition 4.3 (Computer program).
A sequence or set of instructions for a computer to execute is called *computer program*. Moreover, computer programs or components of the latter are called *user interface* or *HMI (human machine interface)* if it allows data to be transferred to and from the computer program.

Algorithm 8 Prototype of simulation

```
Input: Model $\Sigma_M$
Input: Parameter sets $\mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$
Input: Step size $h \in \mathbb{R}^+$
Input: Sampling time $T \in \mathbb{R}^+$
Input: One step method $\Psi$
Input: Terminal time $\overline{T} \in \mathcal{T}$
Input: Error threshold $\overline{\varepsilon} \in \mathbb{R}^+$
Input: Safety factor $\eta \in (0,1)$
1: procedure CLASS SIMULATION($\Sigma_M, \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}, h, T, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
2:     for all $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ do
3:         while $t_0 < \overline{T}$ do
4:             $\mathbf{x}(t_0 + T) \leftarrow$ CLASS SIMULATOR($\Sigma_M, t_0, \mathbf{x}(t_0), h, T, \overline{T}, \Psi, \overline{\varepsilon}, \eta$)
5:             $t_0 \leftarrow t_0 + T$
6:         end while
7:     end for
8: end procedure
Output: Trajectory $\mathbf{x}(\cdot)$
```

Table 4.1: Advantages and disadvantages of splitting simulation/simulator

Advantage	Disadvantage
✓ Separate test from evaluation	✗ Induces structural overhead
✓ Allows parallel runs	✗ Requires parameter separation

Having taken the first step to compute solutions for a given parameter set, our next aim is to analyze the results.

4.2 Processing of results

Based on a simulation, the respective results are used to understand the system behavior and generate predictions for it. To this end, the results are processed to

- design replications,
- estimate performance metrics (e.g., via statistics), and
- analyze system and experiment (e.g., via textual and graphical output).

Remark 4.4
Replication design is an aim to obtain the most (deterministic/statistical) information from simulation runs for least computational cost. To this end, one typically tries to minimize the number of replications/experiments or the length of the latter.
Performance metrics aim to compute point estimates or confidence intervals for system parameters of interest. Here, one focuses on sample size and independence of observations which may be critical in the analysis step.
System analysis and experimentation aim to understand system behavior, generate performance predictions for possibly different scenarios and highlight design trade-offs. Examples range from parametric analysis of inputs or scenarios of operation.

Starting point of processing are the results obtained as data from the simulation. Here, we like to note that these results differ significantly with respect to time, or more accurately with respect to the time considered by the simulation.

Definition 4.5 (Terminating/non-terminating simulation).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. If the terminal time is finite, i.e. $\overline{T} < \infty$, then we refer to Θ as *terminating simulation*. If $\overline{T} = \infty$, then Θ is called *non-terminating simulation*.

The difference between the latter two cases has a major impact on processing of results. In case of a terminating simulation, several simulation runs can be executed sequentially and the results can be combined in an aposteriori manner, i.e. after finishing the runs. For non-terminating simulations, it is not possible to wait for termination but results need to be processed while the simulation is running. Also subsequent runs are not possible but parallel runs are required. Here, we start with the outcome of a simulation:

Definition 4.6 (Replication).
Given a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ with terminal time $\overline{T} < \infty$ and inputs $\{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)_{i \in \mathcal{I}}\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Then the set of outputs $\mathcal{R} := \{(\mathbf{x}_j(t_i))_{i,j \in \mathcal{I}}\}$ is called *replication*.

Remark 4.7
Since a simulation may call several simulators with different parameters, we used the index $\mathbf{x}_j$ to indicate the simulator result and t_i to indicate the simulation time index.

Based on a replication, many different processing steps may be taken. The first and particularly important one is to make a replication accessible. One of the most basic procedures for accessibility is th so called data dump, which is the essence of saving.

Definition 4.8 (Data dump).
Suppose a replication $\mathcal{R}$ to be given by a computer program. Then we call a transfer of the replication to another computer program a *data dump*.

While the latter definition is very general, typical implementations are export functions to store data to a storage device, e.g., as a csv or json file, or to transfer tables from one database to another one using, e.g., SQL. Note that the second computer program may also be the operating system. Hence, a data dump is/should be used upon termination of a (terminating) simulation. While it is also possible to execute a data dump while the simulation is running, it is quite uncommon as it results in unnecessary workload. During runtime of a simulation, other forms of output are used.

Definition 4.9 (Widget).
Given a replication $\mathcal{R}$ within a computer program. Then we call a transfer of the replication to a user interface a *widget*.

Hence, a widget refers to a textual or graphical interface. Note that also a console is a widget. Based on the latter two definitions, we can extend our sketch of a simulation UML diagram, cf. Figure 4.3.

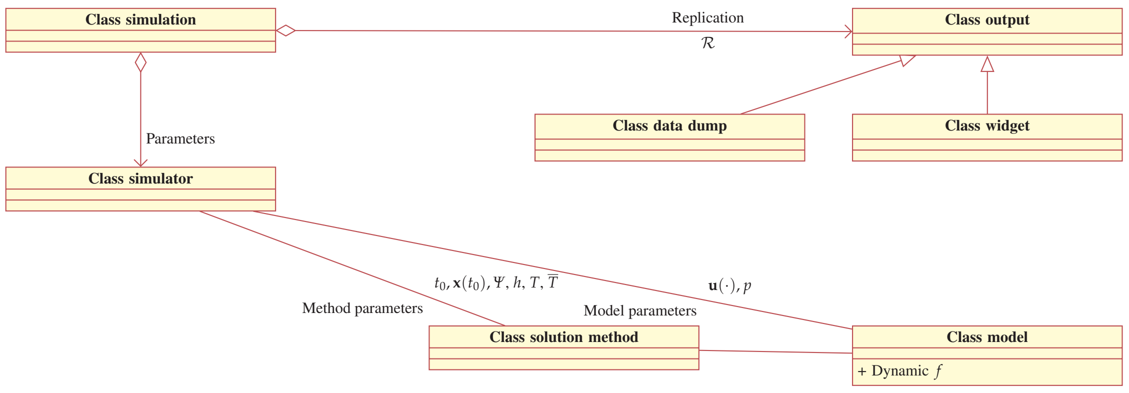


Figure 4.3: Sketch of UML diagram of class simulator with data processing

Apart from accessibility, several other steps for data processing exist:

Definition 4.10 (Data processing).
Consider a replication $\mathcal{R}$ to be given by a computer program. Then possible processing steps include

- Saving — making a replication accessible,
- Validation — ensuring correctness and relevance of data,
- Sorting — arranging data according to an order or in sets,
- Summation — reducing data to their statistical properties,
- Aggregation — combining specific data,
- Classification — separating data by properties, and
- Reporting — listing data or results of processing steps.

The data processing parts are quite fundamental but serve as foundation for so called data analytics used to gain insights from data to support business, social or technical processes.

Definition 4.11 (Data analytics).
Given a replication $\mathcal{R}$ by a computer program, data analytics refers to steps including

- Inspecting — detecting corrupt / inaccurate / incomplete data,
- Cleaning — deleting, replacing, modifying or completing data,
- Transforming — converting data into formats / structures, and
- Modeling — defining and analyzing data requirements.

Within this lecture, we only deal with basic methods from data processing. As indicated by the definitions, both data processing and data analytics are operating on replications and are therefore associated with the output of a simulation. Figure 4.4.
There are several options to sort and report data. The most simple one is a table, cf. Table 4.2.

Table 4.2: Example of a data dump

t	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$

Table 4.2 – continued from previous page

16	−6	−2	11
17	−6	−3	10
18	−6	−3	9
13	−5	−2	12
14	−5	−2	11
15	−5	−2	11
19	−5	−3	8
20	−5	−3	7
21	−5	−3	6
11	−4	−2	13
12	−4	−2	12
22	−4	−3	6
10	−2	−2	13
23	−2	−3	5
24	−2	−3	5
25	−2	−4	6
9	−1	−1	13
3	0	0	8
2	1	2	8
1	2	3	7
4	2	0	10
6	2	−1	12
7	2	−1	12
8	2	−1	13
5	3	0	11
0	5	5	5

By including knowledge such as t denoting time, which can therefore be used as sort key, we can apply the data processing step *sorting* to Table 4.2 and obtain the sorted results given in Table 4.3.

Table 4.3: Sorted data dump from Table 4.2 with sort key t

t	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$
0	5	5	5

Table 4.3 – continued from previous page

1	2	3	7
2	1	2	8
3	0	0	8
4	2	0	10
5	3	0	11
6	2	−1	12
7	2	−1	12
8	2	−1	13
9	−1	−1	13
10	−2	−2	13
11	−4	−2	13
12	−4	−2	12
13	−5	−2	12
14	−5	−2	11
15	−5	−2	11
16	−6	−2	11
17	−6	−3	10
18	−6	−3	9
19	−5	−3	8
20	−5	−3	7
21	−5	−3	6
22	−4	−3	6
23	−2	−3	5
24	−2	−3	5
25	−2	−4	6

The data contained in that table can also be arranged as points in a so called point cloud.

Definition 4.12 (Point cloud).
Given a replication $\mathcal{R}$, a *point cloud* is a widget aligning selected data as dots in a coordinate system.

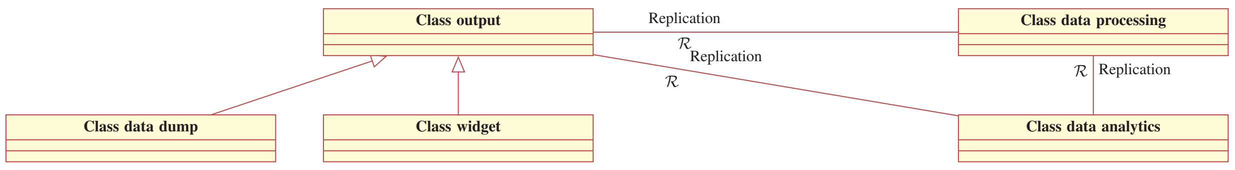


Figure 4.4: Sketch of UML diagram connecting output and data processing/analytics

Task 4.13 (Point cloud)
Use the state coordinates $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2 to plot a point cloud in Carthesian coordinates.*

Solution to Task 4.13: The respective point cloud is given in Figure 4.5.

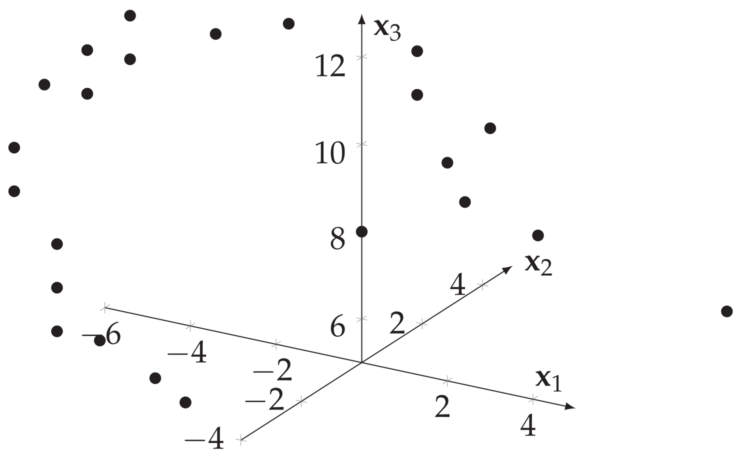


Figure 4.5: Point cloud of three time series from Figure 4.6

While we can already obtain a reporting from both table and point cloud, it does not provide us with any additional information. One step towards aggregation is to include time in the visualization.

Definition 4.14 (Time series / trajectories).
Given a replication $\mathcal{R}$ including time, we call a widget aligning time t to respective data $\mathbf{x}(t)$ a *time series*. Linking points of a time series in timely order is called a *trajectory plot*.

Task 4.15 (Time series)
Use the state coordinates $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2 to plot the time series.*

Remark 4.16
A time series may be given by an entry of the state vector $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_{n_x})$*, but may also correspond to the complete state as an outcome of a simulator* $\mathbf{x}$.

Solution to Task 4.13: The respective time series and trajectory plots are given in Figure 4.6.

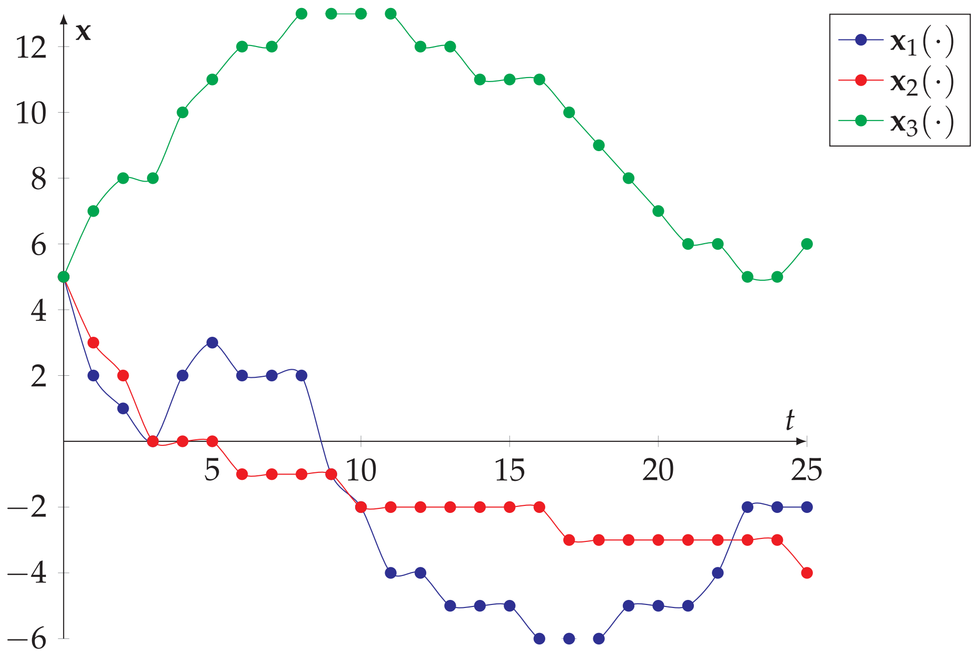


Figure 4.6: Sketch of three time series

Within our example, we observe that the different time series seem to diverge to different levels within their subspace. To analyze the latter in particular with respect to convergence of a time series or its variation, the statistical terms of mean value, sample variance and confidence interval can be applied. The respective information is contained in the aggregation and classification components of data processing.

Definition 4.17 (Mean, sample variance and confidence interval).
Consider a replication $\mathcal{R} = \{(\mathbf{x}_j)_{j \in \mathcal{I}}\}$ to be given. Then we call

$$\mathrm{E}(\mathcal{R}) := \frac{\sum\limits_{j \in \mathcal{I}} \mathbf{x}_j}{\sharp \mathcal{I}} \tag{4.2}$$

mean or *expected value* of $\mathcal{R}$. Moreover, we call

$$\mathrm{S}(\mathcal{R}) := \frac{\sum\limits_{j \in \mathcal{I}} \left(\mathbf{x}_j - \mathrm{E}(\mathcal{R})\right)}{\sharp \mathcal{I} - 1} \tag{4.3}$$

sample variance and

$$\mathrm{I}(\mathcal{R}) := \left[\mathrm{E}(\mathcal{R}) \pm t(\sharp \mathcal{I} - 1, 1 - \alpha/2) \frac{\mathrm{S}(\mathcal{R})}{\sqrt{\sharp \mathcal{I}}}\right] \tag{4.4}$$

confidence interval where $t(\sharp \mathcal{I} - 1, 1 - \alpha/2)$ is the critical value of the t-distribution (or z-distribution) with confidence level $1 - \alpha/2$. We call

$$Q_1(\mathcal{R}) := \mathrm{E}\left(\{\mathbf{x}_j\} \mid \mathbf{x}_j \text{ is in the lowest 25\% of } \mathcal{R}\right) \tag{4.5}$$

$$Q_3(\mathcal{R}) := \mathrm{E}\left(\{\mathbf{x}_j\} \mid \mathbf{x}_j \text{ is in the highest 25\% of } \mathcal{R}\right) \tag{4.6}$$

first and third quartile. Last, we call any element of $\mathcal{R}$ satisfying

$$\mathbf{x}_j \notin [Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)] \tag{4.7}$$

for $k > 0$ an *outlier*.

Remark 4.18
There is no common definition of an outlier. Here, we use the so called Tukey's fence.

The latter components are combined in a so called boxplot.

Definition 4.19 (Statistical plots).
Given a replication $\mathcal{R}$, a *boxplot* is a five number summary containing median, minimum and maximum as lines and first and third quartile marking a box. Outlier may be added as points.

Task 4.20 (Boxplot)
Compute boxplots for the states $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ *from the replication* $\mathcal{R}$ *given in Table 4.2.*

Solution to Task 4.20: The respective boxplots are given in Figure 4.7.

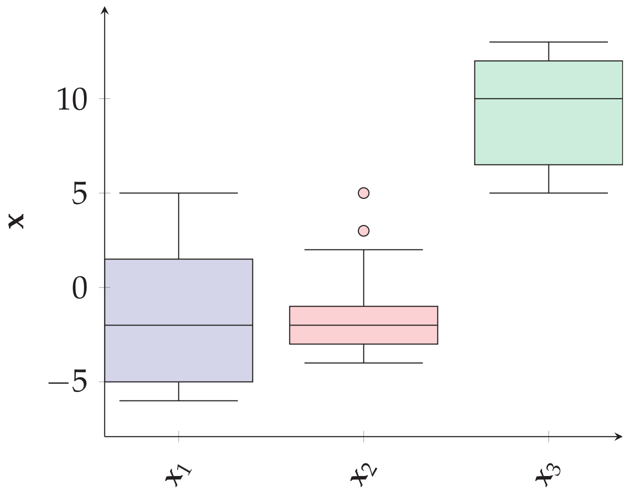


Figure 4.7: Box plot of three time series from Figure 4.6

As a last idea of data processing, we include visualization using a representation of the system at hand. For these cases, the results from the simulation $\mathcal{R}$ must be mapped to factors contained in the graphical models. The advantage of such a visualization is that the implications on reality, and in particular of dependencies of parameters and scenarios is intuitively accessible. Figure 4.8 provides an overview.

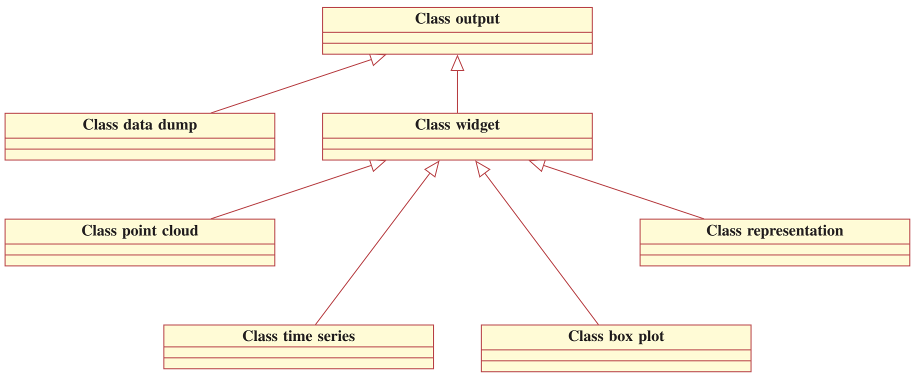


Figure 4.8: Sketch of UML diagram of class output

Unfortunately, the dependencies are in many cases hidden, but can be calculated using the approach of sensitivities, which we address next.

4.3 Sensitivity

In many applications, it is essential to find out how the choice of inputs affects the outputs. To assess this effect analytically, the idea of sensitivities can be applied at least to those inputs, which can be varied continuously.

Definition 4.21 (Sensitivity).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Furthermore suppose the inputs $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ to be fixed and the sets $\mathcal{T}, \mathcal{X}, \mathcal{U}^{\mathbf{T}}$ and $\mathcal{P}$ to be continuous. Then we call

$$\frac{\partial \mathbf{x}_j}{\partial t_0}(t_i), \quad \frac{\partial \mathbf{x}_j}{\partial \mathbf{x}(t_0)}(t_i), \quad \frac{\partial \mathbf{x}_j}{\partial \mathbf{u}(\cdot)}(t_i), \quad \text{and} \quad \frac{\partial \mathbf{x}_j}{\partial p}(t_i) \tag{4.8}$$

sensitivity of the output $\mathbf{x}_j(t_i)$ with respect to the inputs.

The idea of sensitivities is to assess the impact of an input change on the output change, i.e. whether the effect decreases the output, leaves it unchanged, increases it, structurally changes it or bifurcates it.

Task 4.22
Describe cases for input/output changing effects.

Solution to Task 4.22: A decreasing effect is given, e.g., for dampers. Unchanged output refers to independence of output from input. An increase occurs for a gain, e.g., via an electric amplifier. Structural changes may occur, e.g., if parameters affecting poles shift the latter from the negative to the positive complex halfplain. Last, bifurcation may occur if a change of input results in a modification of equilibria and the solution converges to a different equilibrium.

Remark 4.23
In case $\mathcal{T}, \mathcal{X}, \mathcal{U}^{\mathbf{T}}$ and $\mathcal{P}$ are not continuous, there exists no derivative and hence this idea cannot be followed.

By definition, the sensitivity is locked to the chosen values of the inputs $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$, yet it may serve as a pointwise description, i.e. for fixed t_i, or as a field, i.e. along the time series $(t_i)_{i \in \mathcal{I}}$. In practice, we will not compute the derivatives of the outputs but instead use an approximation. To this end, at least two simulation outputs for an input variation are required.

Definition 4.24 (Numerical sensitivity).
Consider a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and suppose two inputs $(t_{0,1}, \mathbf{x}_1(t_{0,1}), \mathbf{u}_1(\cdot), p_1), (t_{0,2}, \mathbf{x}_2(t_{0,2}), \mathbf{u}_2(\cdot), p_2) \in \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ to be given where $\mathcal{T}$, $\mathcal{X}, \mathcal{U}^{\mathbf{T}}$ and $\mathcal{P}$ are continuous. Then we can use

$$\frac{\partial \mathbf{x}_j}{\partial t_0}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{t_{0,1} - t_{0,2}} \tag{4.9}$$

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}(t_0)}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{\mathbf{x}_1(t_{0,1}) - \mathbf{x}_2(t_{0,2})} \tag{4.10}$$

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{u}(\cdot)}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{\mathbf{u}_1(\cdot) - \mathbf{u}_2(\cdot)} \tag{4.11}$$

$$\frac{\partial \mathbf{x}_j}{\partial p}(t_i) \approx \frac{\mathbf{x}_{j,1}(t_i) - \mathbf{x}_{j,2}(t_i)}{p_1 - p_2} \tag{4.12}$$

as an approximation of the sensitivities provided the distances between the elements of respective inputs are small enough and the remaining inputs are equivalent.

Remark 4.25
Note that sensitivities have got nothing to do with statistics, i.e. the results is purely deterministic and not stochastic.

To process sensitivities, one can either apply a data dump or a 2D point cloud. Regarding the latter, the abscissa describes the respective input $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)$ and the ordinate the sensitivities.
Based on the numerical approximation of sensitivities, we now discuss the possible outcomes of such an analysis more closely. In applications, the above mentioned cases are of very different interest. Here, we highlight the implications but the background is beyond the scope of this lecture:

- Decreasing or unchanged output can be seen as robust behavior, yet to some extend allows us to ignore changes regarding these inputs.

- Increasing output indicates that the input change must be limited, yet as long as no structural or bifurcation changes occur may be tolerated.
- Structural changes reflect that properties of the behavior of the system are changed. For simulation, it is of particular interest to find those point of input changes, which form the boundary between two structures.
- Bifurcation changes indicate changes in properties of the system, not its behavior. Similar to structural changes, the switching points are of interest.

Utilizing the latter, we introduce the so called scenario analysis.

Definition 4.26 (Scenario analysis).
A scenario analysis is given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and three inputs reflecting

- the worst case,
- the best case, and
- the base case which reflects the most likely scenario.

Scenario analysis is focused on finding those switching values, which lead to a change from base to worst or to best case, and to identify the tolerable size of deviations.

For both scenario analysis and sensitivity analysis, we can apply our simulation prototype from Algorithm 8. To this end, the parameters $(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)$ are chosen to represent the specifications of the scenarios. Regarding sensitivity, similarly the parameter variations need to be supplied. Such definitions can be automated, which is part of the upcoming Chapter 5.

CHAPTER 5

TESTING AND AUTOMATION

After discussing simulation and data processing of respective results in the previous chapter, we now turn towards validation and verification of the original intention of simulation. To this end, we focus on requirements and respective formulation as well as testing these requirements to complete the V model approach, cf. Figure 5.1.

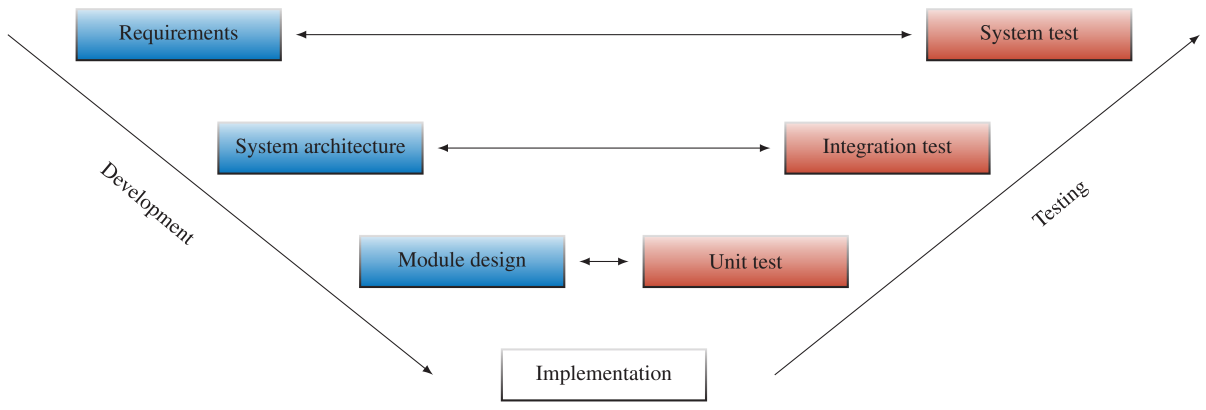


Figure 5.1: V model for system development

We start by a short recap regarding those parts of simulation which we treated in the previous Chapters 3 and Chapter 4. Taking one step backwards, we then introduce the general concept of a requirement including its three parts component, specified condition and quality in Section 5.1. In the following Section 5.2, we introduce and discuss how test cases can be derive to structure the process and aim of testing. To this end, we discuss a prototype test and the concept of grids. Based on the latter, in Section 5.3 we build a hierarchy of tests, which allows us to address integration and system issues. In the last Section 5.4, we additionally address automation of testing efforts and how these can be structured efficiently.

5.1 Requirements and testing

Previously, we considered

- differential equation solvers and models in Chapter 3, as well as
- simulator, simulation and data processing in Chapter 4

and encapsulated these algorithms in modules which led us to the system architecture sketched in Figure 5.2.

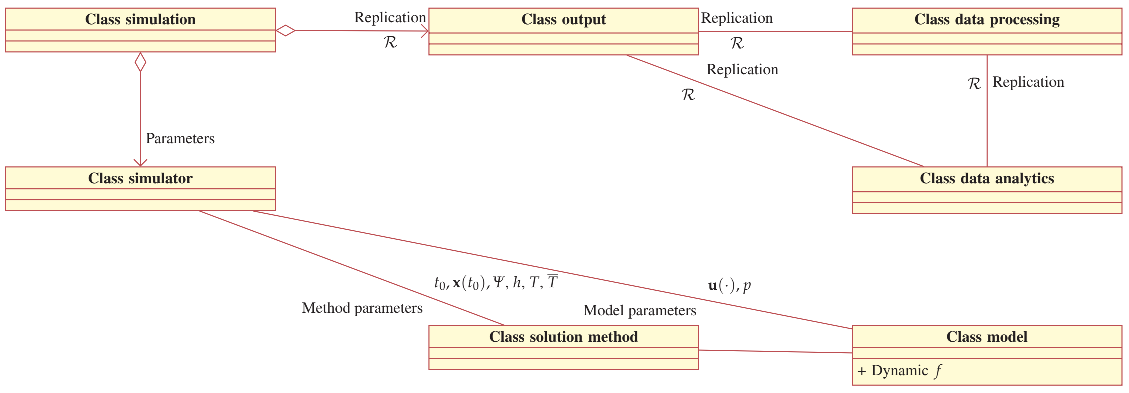


Figure 5.2: Sketch of UML diagram of class simulator with data processing

To complete Figure 5.1, we need to define what the terms requirements and test mean. Within this lecture, we follow ISO 9000:2005 [12], ISO/IEC DIS 25000:2014 [11], ISO 29148:2011 [9] and subsequent norms to introduce necessary terms. For further details, we refer to [9, 11, 12]. The very basic term is the so called need or in our case information need:

> An (information) need is an insight necessary to manage objectives, goals, risks, and problems.
>
> ISO/IEC 15939 (2007)

Within this lecture, the distinction between objectives, goals, risks and problems are outside our scope. Instead, we generically consider a need to be without intent. Utilizing the term of a need, a requirement can be formulated:

> A requirement is a need or expectation that is stated, generally implied or obligatory.

ISO 9000 (2005)

Remark 5.1
We like to stress that the different between a need and an expectation is that for a need the insight is necessary while for an expectation this is not the case. Hence every need is an expectation but not every expectation is a need. Moreover, an expectation is implied if it is NOT stated but still an insight. And last, an expectation is obligatory if it is implied and necessary, i.e. only needs may be obligatory.

The term requirement in ISO 9000 is not specific as to how such a requirement shall be formulated. For this reason, the range of requirement formulations found even for one task may be very large. In many cases, requirements are formulated as patterns/phrases in sentences or user stories.

Task 5.2
Consider a system Σ to be given. Check whether

1. *The system Σ is described by a model Σ_M.*
2. *The model Σ_M can be evaluated using a simulator Φ.*

are requirements for the system.

Solution to Task 5.2: For 1, an expectation for system Σ is given by „description by a model Σ_M“. Regarding 2, the expectation is not linked to the system and is therefore not a requirement of the system.

From Task 5.2(2) we can additionally observe that the formulation is not exact. For a given simulator Φ, the assertion „can be evaluated“ either holds true or holds false. Yet, if it holds false, there may still exist another simulator such that the assertion holds true. For this reason, it makes sense to use patterns/phrases to obtain unique questions/answers.
Moreover we observe from Task 5.2 that an answer for each requirement must be derived. In this context, the answer is called quality. Here, we use the reference object of a component, which offers us the freedom to choose whether we consider a single function, an integration step or the overall system.

Definition 5.3 (Component).
Consider the system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Then we call any subset of this architecture $\gamma : \mathcal{U} \to \mathcal{Y}$ with input set $\mathcal{U}$ and output set $\mathcal{Y}$ a *component*.

Note that while technically not necessary, in practice components are typically defined as connected subsets within the architecture. Based on these components, we can transfer the concept of quality.

> Quality is the capability of a component to satisfy stated and implied needs when used under specified conditions.
>
> ISO/IEC DIS 25000 (2014)

Transferred to our setting in simulation, we can define the following:

Definition 5.4 (Quality).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Then we call $Q : \mathcal{Y} \to \mathbb{R}_0^+$ given by

$$Q(\gamma(\mathbf{u})) = Q(\mathbf{y}) \tag{5.1}$$

quality of a component.

Definition 5.5 (Specified conditions).
Given a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ and a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of Θ. Then we call $\Gamma := \{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$ *test set* or *specified conditions*.

In particular, we define the following:

Definition 5.6 (Requirement).
Given a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$ together with quality criteria $Q : \mathcal{Y} \to \mathbb{R}_0^+$ and specified conditions $\Gamma := \{(t_0, \mathbf{x}(t_0), \mathbf{u}(\cdot), p)\} \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Then we call the triple $R := (\gamma, Q, \Gamma)$ a *requirement*.

As outlined before, requirements are often formulated verbally. Still, the quality criteria defined in Definition 5.4 together with the specified conditions from Definition 5.5 are considered to uphold certain properties [9].

Assumption 5.7 (Properties of quality criteria and specified conditions)
A requirement shall be

- complete — each requirement must fully describe the required and to-be-delivered functionality
- consist – a requirement must not contradict itself and other requirement
- agreed – all stakeholders accept the validity of a requirement
- unambiguous – a requirement is interpreted only one way by all readers
- verifiable – a requirement can be proven by a test or measurement
- traceable – the origin of the requirement, its implementation and the relationship to other documents can be retraced
- necessary – in the event of failure, a requirement shall cause a deficit
- understandable – a requirement contains only terms with fixed meaning
- feasible — a developer can point out specific facts regarding implementation and costs of a requirement

Within our system architecture from Figure 5.2, a typical component is the class model and a typical test set are initial conditions for the model/simulation.

Remark 5.8
We like to stress that quality is restricted to stated and implied needs, which is only a subset of requirements for a system. Note that quality as an assertion is linked the concept of a key performance indicator (cf. Definition 2.16), yet it focuses on yes/no answers.

While normatively restricted to needs, the concept of quality can also be applied to requirements. To evaluate the quality of a component regarding a requirement, we need to obtain data/measurements from it.

> A measurement is set of operations having the object of determining a value of a measure.
>
> ISO/IEC 15939 (2007)

Note that implicitly, we already utilized measurement of a component in Definition 5.4 by considering $\mathbf{y} = \gamma(\mathbf{u})$. More formally:

Definition 5.9 (Measurement).
Given a component $\gamma : \mathcal{U} \to \mathcal{Y}$ we call $\mathbf{y} \in \mathcal{Y}$ given by

$$\mathbf{y} = \gamma(\mathbf{u}) \tag{5.2}$$

measurement of the component.

Task 5.10
State a measurement of a Runge-Kutta and a one-step method.

Solution to Task 5.10: The output of the Runge-Kutta method, i.e. endpoint of trajectory $\mathbf{x}(\overline{T})$, is the measurement obtained by executing Algorithm 1 or more generically Algorithm 5 for a one-step method.

Based on measurements, we can describe a so called evaluation method:

> An evaluation method is procedure describing actions to be performed by the evaluator in order to obtain results for the specified measurement applied to component.
>
> ISO/IEC DIS 25000 (2014)

Last, we introduce the so called quality evaluation.

> A quality evaluation is a systematic examination of the extent to which a component is capable of satisfying stated and implied needs.
>
> ISO/IEC DIS 25000 (2014)

Hence, to assess a component, we can apply a quality evaluation. From this, we obtain the following conclusion.

Corollary 5.11 (Test).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. Furthermore suppose specified conditions Γ to be fixed and the quality $Q : \mathcal{Y} \to \mathbb{R}_0^+$ to be given. Then the tripel (γ, Γ, Q) is called a test.

The connection of all terms including the consequences for defining a test are illustrated in Figure 5.3.

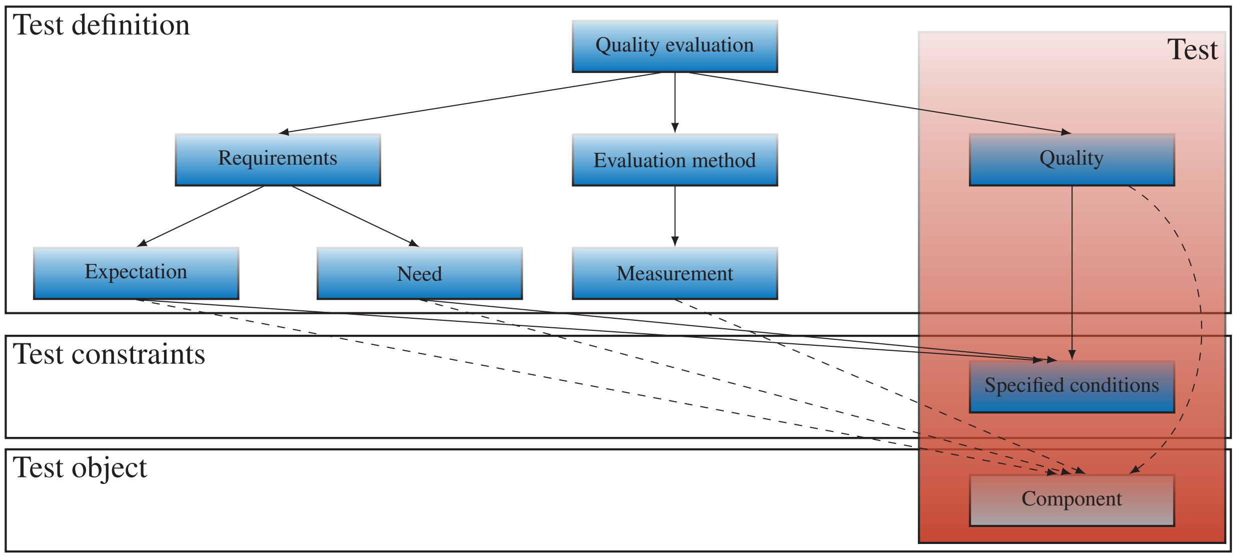


Figure 5.3: Connections of requirement terms

Coming back to the right hand side of Figure 5.1, we distinguish between unit test, integration tests and system tests. To capture the meaning of the latter, we introduce the following:

Definition 5.12 (Unit).
Consider a component $\gamma : \mathcal{U} \to \mathcal{Y}$ as subset of a system architecture given by a simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$. If there exists no component $\gamma_2 \subsetneq \gamma$, then γ is called a *unit*.

Based on units, we can define:

Definition 5.13 (Unit/integration/system test).
Consider a test (γ, Γ, Q).

- If γ is a unit, then the test is called *unit test*.
- If $\gamma = \gamma_1 \cup \gamma_2$ where γ_1, γ_2 are units with $\gamma_1 \subsetneq \gamma$, $\gamma_2 \subsetneq \gamma$, then the test is called *integration test*.
- If $\gamma = \Theta$, then the test is called *system test*.

At this point, we want to highlight the following distinction regarding the outcome of tests, cf. ISO 9000 (2005):

- Test confirming (and providing objective evidence) that the requirements for a specific intended use or application have been fulfilled, are called *validation*.
- Verification is confirmation (including objective evidence) that specified requirements have been fulfilled.

Hence, in order to verify a system, it may be necessary to extend test cases derived from intended use or application to unintended use or misuse. This leads us to the derivation of test cases.

5.2 Derivation of test cases

To obtain test cases, we start on a unit level. Recalling the definition of a unit, we see that it operates on inputs $\mathbf{u} \in \mathcal{U}$ providing outputs $\mathbf{y} \in \mathcal{Y}$. Hence, a unit test is nothing else than checking whether or not for all inputs the respective outputs are correct. To do so, one should follow certain steps:

- Identify units: Units may be functions, methods, classes, or even smaller components depending on the chosen granularity.
- Understand requirements: Making needs and expectations clear helps to define the scope and purpose of a unit test.
- Define test: For each requirement, there should be exactly one test with one test set and one quality.
- Determine inputs/outputs: Collecting information on the unit help to limit the ranges of inputs and outputs.

- Write test code: Utilizing testing frameworks allows to efficiently test units.
- Execute tests: Run to validate correctness of unit.
- Analyze result: The outcome allows to check validity and identify errors/failures to investigate causes.
- Refine: Make the step towards verification.
- Maintain coverage: Unit test reflect standard behavior checks as the code gets larger.

Unfortunately, the number of possible combinations regarding specified conditions Γ may be infinite even if it only a number $\mathbf{u} \in [0,1] \subset \mathbb{R}$. To limit the specified conditions, grids can be applied.

Definition 5.14 (Grid).
Consider specified conditions $\Gamma \subset \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P}$. Suppose $n_{\mathcal{T}}, n_{\mathcal{X}}, n_{\mathcal{U}^{\mathbf{T}}}, n_{\mathcal{P}} \in \mathbb{N}$ to be number of sampling points for each set. Then we call

$$\Gamma := \mathbb{T} \times \mathbb{X} \times \mathbb{U} \times \mathbb{P} \tag{5.3}$$

grid where

$$\begin{aligned}
\mathbb{T} &:= \{t_i \in \mathcal{T} \mid i = 1, \dots, n_{\mathcal{T}}\} \\
\mathbb{X} &:= \{\mathbf{x}_i \in \mathcal{X} \mid i = 1, \dots, n_{\mathcal{X}}\} \\
\mathbb{U} &:= \{\mathbf{u}_i \in \mathcal{U}^{\mathbf{T}} \mid i = 1, \dots, n_{\mathcal{U}^{\mathbf{T}}}\} \\
\mathbb{P} &:= \{p_i \in \mathcal{P} \mid i = 1, \dots, n_{\mathcal{P}}\}
\end{aligned}$$

are grids on the specified conditions.

A prototype of a test considering a grid of specified conditions is given by Algorithm 9.
We observe that the test prototype works similar to the simulation prototype by running all specified conditions (respectively parameters).

Remark 5.15
If the specified conditions are given by a finite set, then Algorithm 9 can also be used for full enumeration.

Algorithm 9 Prototype of test

```
Input: Component $\gamma$
Input: Specified condition $\Gamma$
Input: Quality $Q$
1: procedure CLASS TEST($\gamma, \Gamma, Q$)
2:     for all $\mathbf{u} \in \Gamma$ do
3:         $Q(\mathbf{u}) \leftarrow Q\left(\text{COMPONENT } \gamma(\mathbf{u})\right)$
4:     end for
5: end procedure
Output: Quality $Q(\cdot)$
```

On unit level, many tests are required to check whether very simple function such as GETVARIABLE() or SETVARIABLE() or initializations of storage/objects are working as intended. Regarding such functions, the test description (γ, Γ, Q) is much simpler. Still, these tests form the basis for the so called *continuous integration* and *continuous development*.

5.3 Continuous integration and delivery

Continuous Integration, also termed CI, is a software development practice that involves regularly integrating code changes from multiple developers into a shared repository. The primary goal of continuous integration is to catch integration issues early in the development process, ensuring that the software remains in a releasable state at all times.
Before continuing to dive deeper into continuous integration, we need to clarify the already indicated vocabulary:

Definition 5.16 (Shared repository / version control system).
A *repository* is a storage location, which manages and tracks changes to files over time, provides shared access, collaboration and the option to revert to previous versions if needed. The management tool to work with a shared repository is called *version control system*.

Regarding continuous integration, Git allows to use respective tools implementing the principles and practices of continuous integration. To this end, we define the term releasable as follows.

Definition 5.17 (Releasable).
We call a component $\gamma : \mathcal{U} \to \mathcal{Y}$ *releasable* if all related tests (γ, Γ, Q) satisfy the quality thresholds

$$Q(\gamma(\mathbf{u})) \geq \underline{Q} \qquad \forall \mathbf{u} \in \Gamma. \tag{5.4}$$

To ensure the latter, continuous integration incorporates the following principles:

Definition 5.18 (Principles of continuous integration).
Aiming at code stability through revision loops while allowing for collaboration, continuous integration follows the principles of

- Maintaining a single source repository: Continuous integration emphasizes the use of a single, central version control repository where all developers commit their changes. This allows for easy and frequent integration of code changes and ensures that everyone is working with the latest version of the codebase.
- Automating the build process: The build process, which includes compiling the code and generating necessary artifacts, can be automated, e.g., via commit triggers. This ensures that the code is consistently built and reduces the chance of human error.
- Build in a clean environment: To avoid unexpected issues caused by inconsistent development environments, the build process should be performed in a clean and controlled environment. This helps to eliminate dependencies and ensures reproducibility.
- Keep builds fast: Obtaining a quick feedback on their changes quickly, developers can catch integration issues and errors early, enabling prompt resolution.
- Run automated tests: Automated testing is critical and includes unit tests, integration tests, and regression tests. These tests verify the correctness and quality of the codebase and help catch issues before they propagate further.
- Fail fast: If a build or test fails, it should be identified and communicated to promptly fix it.
- Provide feedback: Feedback helps developers to understand the impact of their changes and facilitates collaboration within the team. It enables quick identification and resolution of issues.
- Practice continuous integration: Regularly integrating changes helps to identify integration issues early and prevents code branches from diverging too much, reducing the effort required to merge them later.

As we can already see from the latter definition, continuous integration considers integration tests rather than unit tests. To perform an integration test, it must be assured that the integrated units are working properly. This results in a hierarchy of test as depicted in Figure 5.4.

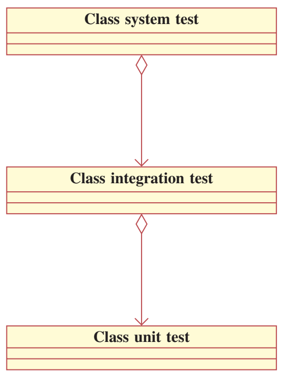


Figure 5.4: Sketch of UML diagram of test classes

Task 5.19
Apply the hierarchy of tests to the simulation setting from Figure 5.2 without output.

Solution to Task 5.19: We obtain the unit/integration/system tests sketched in Figure 5.5.

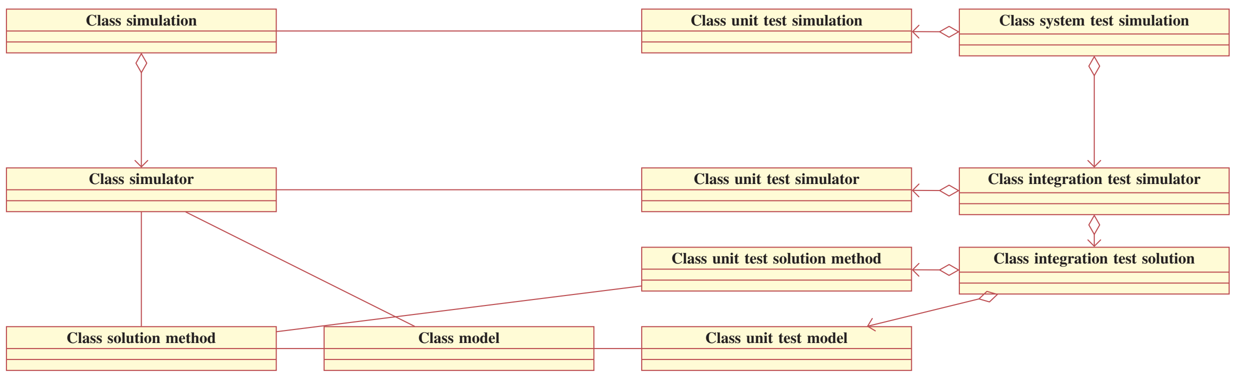

Class simulation
Class unit test simulation
Class system test simulation
Class simulator
Class unit test simulator
Class integration test simulator
Class unit test solution method
Class integration test solution
Class solution method
Class model
Class unit test model

Figure 5.5: Sketch of UML diagram of class simulator with testing

The second part of the section headline sketches the handover from development to operations and is referred to as continuous delivery or continuous deployment. As we have seen before, the requirement for deployment in the DevOps cycle is releasability. The latter in turn requires that all related tests are passed, which leads us to the term test-ready.

Definition 5.20 (Test ready).
We call a component $\gamma : \mathcal{U} \to \mathcal{Y}$ *test ready* if the tests (γ, Γ, Q) cover needs and expectations and are practicable.

As we can see, there are two conditions, practicability and coverage. Here, practicability refers to ability of executing tests within given constraints such as time, resources, and budget. It involves assessing whether the testing objectives can be realistically achieved and whether the testing approach is viable in the given context. Coverage on the other hand refers to the extent to which unit is exercised by a set of tests. It quantifies the amount of code, functionalities, or requirements that have been tested and helps assess the thoroughness and effectiveness of the testing process.

5.4 Automation of testing

Last, we can observe that based on the test ready property all required tests for establishing releasability are given. For such a situation, these tests can be summarized.

Algorithm 10 Automation of test

```
Input: Simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \to \mathcal{X}^{\mathbf{T}}$
Input: Tests $\{(\gamma_i, \Gamma_i, Q_i)\}_{i=1,\ldots,n_\gamma}$
1: procedure CLASS AUTOMATED TEST($\Theta, \{(\gamma_i, \Gamma_i, Q_i)\}_{i=1,\ldots,n_\gamma}$)
2:     for $i = 1, \ldots, n_\gamma$ do
3:         $Q(\mathbf{u}_i) \leftarrow$ CLASS TEST($\gamma_i, \Gamma_i, Q_i$)
4:     end for
5: end procedure
Output: Quality $Q(\cdot)$
```

Remark 5.21
Note that as indicated in Figure 5.5, each integration test aggregates unit tests and completes them with integration test sets, and each system test aggregates integration tests complemented by linking system test sets.

We like to highlight that Algorithm 10 proceeds in an unstructured way and simply executes test by test. Still, if the automation is test ready, then the result of the test allows us to specify whether or not the simulation is releasable. This result is of particular importance in the context of continuous integration and continuous delivery. Here, tools such as GitLab and automations allow us to check whether or not developers have to continue improving the simulation or if the simulation can be delivered to operations in a release.
To improve the automation, it typically makes sense to distinguish between unit, integration and system tests. A possible structure is outline in Algorithm 11.
Within Algorithm 11 we used several ideas to speed up the test for releasability:

- To avoid unnecessary tests, we included the test fail criterion. Hence, if a test is not passed,

Algorithm 11 Automation of tests using test structures

```
Input: Simulation $\Theta : \mathcal{T} \times \mathcal{T} \times \mathcal{X} \times \mathcal{U}^{\mathbf{T}} \times \mathcal{P} \rightarrow \mathcal{X}^{\mathbf{T}}$
Input: Tests $\{(\gamma_i, \Gamma_i, Q_i)\}_{i=1,\dots,n_\gamma}$
 1: procedure CLASS AUTOMATED STRUCTURED TEST($\Theta, \{(\gamma_i, \Gamma_i, Q_i)\}_{i=1,\dots,n_\gamma}$)
 2:     for $i = 1, \dots, n_\gamma$ do
 3:         if $\gamma_i$ is unit then
 4:             $S_{\text{unit}} \leftarrow \{(\gamma_i, \Gamma_i, Q_i)\}$
 5:         else if $\gamma_i$ is integration then
 6:             $S_{\text{integration}} \leftarrow \{(\gamma_i, \Gamma_i, Q_i)\}$
 7:         else
 8:             $S_{\text{system}} \leftarrow \{(\gamma_i, \Gamma_i, Q_i)\}$
 9:         end if
10:     end for
11:     for all $(\gamma_i, \Gamma_i, Q_i) \in S_{\text{unit}}, S_{\text{integration}}, S_{\text{system}}$ do
12:         $Q(\mathbf{u}_i) \leftarrow$ CLASS TEST$(\gamma_i, \Gamma_i, Q_i)$
13:         if $Q(\mathbf{u}_i) < \underline{Q(\mathbf{u}_i)}$ then return
14:         end if
15:     end for
16: end procedure
Output: Quality $Q(\cdot)$
```

then the algorithm will automatically terminate and not consider the remainder of tests as in Algorithm 10.

- The tests are structured to consider unit tests first. The idea behind this sequencing is that if a unit test is not passed, then it does not make sense to perform the integration test or system test.

Combined, automation of the continuous integration and continuous delivery process allows us to structure the testing and the obtained errors. This allows us to validate whether the simulation satisfies the requirements and verify that no relevant empty testing areas exist. As a result, we can assume that the simulation shows the required quality to be released.

BIBLIOGRAPHY

[1] BULIRSCH, R. ; STOER, J.: *Numerische Mathematik 2*. Springer, 2005

[2] CHACON, S. ; STRAUB, B.: *Pro Git*. Apress, 2022

[3] COWELL, C. ; LOTZ, N. ; TIMERLAKE, C.: *Automating DevOps with GitLab CI/CD Pipelines*. Packt Publishing, 2023

[4] DEUFLHARD, P. ; BORNEMANN, F.: *Scientific computing with ordinary differential equations*. Springer, 2012

[5] DEUTSCHES INSTITUT FÜR NORMUNG E.V.: *Internationales Elektrotechnisches Wörterbuch Teil 351: Leittechnik (DIN IEC 60050-351:2014-09)*. Beuth, 2014

[6] FARLEY, D.: *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, 2022

[7] GLÖCKLER, M.: *Simulation mechatronischer Systeme: Grundlagen und technische Anwendung*. Springer, 2014

[8] IGLBERGER, K.: *C++ Software Design: Design Principles and Patterns for High-Quality Software*. O'Reilly Media, 2022

[9] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Systems and software engineering – Life cycle processes – Requirements engineering (ISO 29148:2011)*. ISO, 2011

[10] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Automation systems and integration – Key performance indicators (KPIs) for manufacturing operations management (ISO 22400:2014)*. ISO, 2014

[11] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE (ISO/IEC 25000:2014)*. ISO, 2014

[12] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Quality management systems — Fundamentals and vocabulary (ISO 9000:2015)*. ISO, 2015

[13] KHALIL, H.K.: *Nonlinear Systems*. Prentice Hall, 2002

[14] LUNZE, J.: *Automatisierungstechnik*. 5th edition. DeGruyter, 2020

[15] SONTAG, E.D.: *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, 1998

[16] VEREIN DEUTSCHER INGENIEURE E.V.: *VDI/VDE 2206 „Entwicklung mechatronischer und cyber-physischer Systeme“ (VDI/VDE 2206:2019)*. VDI, 2019

[17] WILKE, C.O.: *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O’Reilly Media, 2019

[18] ZIEGLER, B.P. ; MUZY, A. ; KOFMAN, E.: *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. Academic press, 2018

Jürgen Pannek
Institute for Intermodal Transport and Logistic Systems
Hermann-Blenck-Str. 42
38519 Braunschweig

During summer term 2024 I give the lecture to the module *Simulation of mechatronic systems (Simulation mechatronischer Systeme)* at the Technical University of Braunschweig. To structure the lecture and support my students in their learning process, I prepared these lecture notes.
The aim of the module is to classify simulation techniques from numerical mathematics and apply these to mechatronic case studies. After completing the module, the students shall be able to recall, categorize, apply, select and rate simulation methods to mechatronic use cases. Moreover, students shall be able to describe, explain, evaluate, analyze and assess simulation results. As such, students shall be capable to derive and apply automation procedures for deployment, simulation and testing of digital models.