

Testing Challenges for Cyber-Physical Systems

TNO 2025 R11688 – 12 October 2026

Testing Challenges for Cyber-Physical Systems

Author(s)	Jan Tretmans
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	38 (excl. front and back cover)
Number of appendices	0

All rights reserved

No part of this publication may be reproduced and/or published by print, photoprint, microfilm or any other means without the previous written consent of TNO.

Acknowledgements

Valuable feedback and suggestions were provided by Benny Akesson, Leonardo Barbini, Deb-jyoti Bera, Petra van den Bos (University of Twente), Jos Hegge, Marco Vicari, Jacco Wesselius, and Tim Willemse (Eindhoven University of Technology). The research described in this report is supported by the Netherlands Organisation for Applied Scientific Research TNO.

©2026 TNO

Summary

Context and Motivation Cyber-Physical Systems (CPS), such as autonomous vehicles, semiconductor equipment, industrial printers, medical devices, and smart infrastructure, increasingly rely on complex software for their operation, control, and communication. Consequently, system quality, reliability, safety, and dependability are ever more determined by software quality. However, software testing, as a means to assess quality, is struggling to keep pace with the rapid growth in size, complexity, connectivity, and evolution of software. More and more, testing is becoming the bottleneck to innovation. This report analyses the trends and challenges that threaten the testability of future generations of CPS software and it identifies research directions to mitigate these threats.

Testing Challenges

- *Multi-disciplinarity.* CPS integrate software with physical components, which requires combining expertises, methodologies, and models from different disciplines for testing. Use of simulations and digital twins is necessary but introduces its own validation challenges.
- *Size and complexity.* Software size, number of components, configuration parameters, and interdependencies grow faster than testing can handle, leading to combinatorial explosion.
- *Variability.* CPS are delivered in many variants and configurations. While product-line engineering mitigates development effort, testing must still be performed on concrete instances, leading to another combinatorial explosion in testing.
- *Evolution.* Frequent changes in software demand fast and selective re-testing, but methods for test impact analysis and regression test selection are still immature.
- *Component-based development and reuse.* Systems combine in-house software with legacy, outsourced, open-source, and third-party components. Determining what, how much, and at which level to re-test components and integrations is a major challenge.
- *Connectivity.* CPS are connected and increasingly operate together, e.g., as a system-of-systems. Limited central control, independent management, missing global specifications, dynamic composition, and runtime changes necessitate testing beyond development time.
- *Beyond functionality.* Non-functional quality characteristics such as safety, security, reliability, usability, and availability are often non-compositional and can only be tested at higher system levels, where testing is expensive and more difficult to automate. New quality characteristics such as sustainability and fairness (of AI systems) emerge for which new testing methodologies have to be developed.
- *Environment.* The importance of the CPS environment increases, e.g., for safety and security testing. Testing exhaustively for all possible environments is infeasible, so selection, modelling, and simulation of realistic environmental behaviour is necessary, making environment validity an issue.
- *Climbing the test pyramid.* Testing CPS shifts from unit testing to higher integration and system levels, which is harder to automate and more expensive. Moreover, it requires

abstraction that introduces uncertainty and nondeterminism. Test effort shall be balanced across the test pyramid to optimize cost, coverage, and confidence.

- *Agile and test automation.* Large automated test suites are hard to maintain, suffer from redundancy, and may give a false feeling of confidence (“pesticide paradox”). Higher-level testing often lacks stable specifications in agile contexts.
- *Runtime testing.* For autonomous, self-adaptive, and continuously evolving systems, complete pre-release testing is impossible. This creates a need for after-release, runtime testing and quality control, beyond passive monitoring, and the challenge is how to do that.
- *AI.* Artificial intelligence is a vibrant and dynamic area with many new developments and breakthroughs happening in quick succession. This leads to many questions, challenges, and unclarity for testing and AI: how can AI be used for testing, how shall systems developed using AI be tested, and how shall systems that contain AI be tested.
- *Microservices.* Testing an individual microservice can be done with traditional testing methods. While testing a collection of them, many of the above trends and challenges come together, in particular, shifting testing efforts towards higher levels in the pyramid.

Research Directions

- *Testability-by-design:* Make testability an explicit architectural quality attribute, with testers involved early in system design.
- *Model-centric testing:* Develop an integrated approach to modelling, including model-based system development (MBSE), model-based testing (MBT), digital twins, and environmental modelling, and use models as the basis for testing.
- *Compositional testing:* Systematically decompose testing across components, features, and versions, in an attempt to mitigate combinatorial explosion.
- *Runtime testing:* Extend testing into operation through active, non-intrusive runtime testing and quality control.
- *AI and testing:* Investigate methods for testing systems containing AI as well as for using AI to improve test efficiency and effectiveness.
- *Microservices as testbeds:* Use microservice architectures as structured case studies for advancing scalable testing methods.

Concluding Testing remains essential for CPS quality, however, current testing approaches do not scale to future generations of CPS, thus likely hampering innovation. Research and development for new testing methods are needed.

Contents

Summary	3
Contents	5
1 Introduction.....	6
2 Scope	7
2.1 Cyber-Physical Systems	7
2.2 Testing	7
3 Challenges	12
3.1 Multi-Disciplinarity	12
3.2 Size and Complexity	13
3.3 Variability.....	14
3.4 Evolution.....	15
3.5 Components.....	15
3.6 Connectivity	17
3.7 Quality Characteristics	17
3.8 Environment.....	18
3.9 Climbing the Pyramid.....	20
3.10 Agile and Test Automation	21
3.11 Runtime Testing.....	21
3.12 AI and Testing	22
3.13 Microservices.....	24
4 Research Directions.....	27
4.1 Testability	27
4.2 Modelling.....	27
4.3 Compositionality	29
4.4 Runtime Testing.....	30
4.5 AI and Testing	30
4.6 Microservices.....	30
4.7 More Research Topics	31
5 Concluding Remarks	33
References.....	34

1 Introduction

Complex, high-tech, cyber-physical systems (CPS) increasingly depend on software: software controls, connects, and monitors almost every aspect of system operation, be it a printer, a wafer-lithography system, an airplane, a 'smart' factory, or a car [1]. Consequently, the quality of CPS is ever more determined by the quality of the software in such systems, or, the other way around, system failures are increasingly due to software bugs. Moreover, we build ever larger, more complex, and increasingly connected software systems, with more and more options and features, evolving ever faster, building on and reusing legacy, out-sourced, and open-source software, while using new, sophisticated software architectures, development techniques, and paradigms [2, 3, 4, 5, 6, 7].

Systematic testing is one of the most important ways of assessing the quality of the software in CPS [8, 9, 10]. There is, however, a serious threat that software testing techniques and quality assessing methodologies cannot keep pace with software development, and this will endanger the fast and thorough testing of future generations of CPS, thus hampering innovation.

In this report, trends, challenges, and research directions are identified for testing of future generations of CPS, with emphasis on the software part of CPS. These trends, challenges, and possible research directions come from discussions, observations, presentations, and conversations with many people, including TNO-ESI partners, involved in testing in various roles, such as manager, tester, project coordinator, test automation engineer, and researcher. Together, they present an outlook on issues that can be expected when testing CPS, and a vision and research agenda for mitigating these issues.

Overview Chapter 2 starts with setting the scope by introducing the kind of systems and testing that are considered. Chapter 3 discusses trends in CPS developments, and related challenges of testing future generations of CPS systems. Chapter 4 discusses possibilities for research in this domain, after which conclusions are presented in Chapter 5.

2 Scope

This report is about “testing challenges for cyber-physical systems”. In this chapter, the main concepts, ‘cyber-physical systems’ and ‘testing’, are introduced.

2.1 Cyber-Physical Systems

Cyber-physical systems (CPS) are systems in which a computational part (‘cyber’, or software part) and a physical part (mechanical, optical, electrical, mechatronic, ...) work tightly together to provide the functionalities of the system [11, 12, 13, 14]. The software part typically takes care of control, networking, communication with the outside world, user interfacing, monitoring, data processing and analytics, processing sensor data, controlling actuators, etc. The physical parts perform the physical processes such as dealing with movement, energy, engines, light, force, and physical objects. The physical and software elements in a CPS are closely connected, integrating the physical and cyber worlds, to make the entire system perform its tasks.

Designing, developing, and testing a CPS requires multi-disciplinary approaches, merging, on the one hand, physical engineering such as mechanics, mechatronics, electronics, and optics, and, on the other hand, software engineering, including data processing, networking, and control. These different approaches require different and non-matching theories, formalisms, methods, and tools to deal with the cyber and physical elements, respectively. This makes reasoning, as well as interoperability and integration of the different parts of a CPS challenging. Also compliance, certification, and standards differ for the physical and cyber worlds.

Examples of CPS are manifold: autonomous vehicles, healthcare devices, semiconductor manufacturing equipment, smart building control, industrial control systems, smart energy grid, food processing systems, agricultural robots, traffic management, electron microscopes, robotized warehousing, industrial printers, water management, robots, manufacturing lines, consumer appliances, and many more.

In this report, we consider challenges of testing current and future generations of CPS. In such CPS, the software acts as the CPS ‘brain’, i.e., the software part controls, connects, and monitors many aspects of CPS system behaviour. In addition, innovations in CPS often occur in the software part, leading to a constant increase in size and complexity of the software part. A consequence is that the software ever more determines the quality characteristics of CPS, such as functional correctness, performance, dependability, reliability, and availability [1]. The other way around, failures are more and more often caused by software bugs. Therefore, the emphasis in this report is on testing the ‘cyber’-part of such systems, that is, we consider mainly software testing in the context of CPS.

2.2 Testing

Systematic testing is one of the most important and widely used techniques to check the quality of systems and software. Testing, however, is often a manual and laborious process, that is time consuming and costly. Studies indicate that testing can account for 20% to as much as 70% of the total development effort, depending on the kind of software [15, 16,

17]. Testing is ever more the bottleneck in delivering new versions and upgrades for old systems and in producing new systems, thus hampering innovation. This counts in particular for CPS.

Software Testing Testing is a way of determining the quality of some object, product, system, or service. The object being tested is referred to as the system under test (SUT). This report concentrates on testing of software [18, 19, 20, 16, 21]. We use the following description of ‘testing’:

Testing is a technical process,
performed by executing or experimenting with an SUT,
in a controlled environment, following a specified procedure,
with the intent of measuring one or more quality characteristics of the SUT,
by demonstrating the deviation of the actual status of the SUT
from the desired or required status.

According to this description, testing consists of performing some experiment on the real, working and executing SUT. The experiment is explicitly and systematically designed for the testing of the SUT (a specified procedure), and the experiment is performed in a test environment (controlled environment).

Some properties of the SUT, related to a quality characteristic, are checked. A quality characteristic defines a class of product properties that contribute to the product’s quality. A product quality model defines a set of quality characteristics. For software systems, the current, commonly used quality model is ISO/IEC 25010-2023 [22] that defines 9 main quality characteristics (which can be further divided into sub-characteristics): functional suitability (functionality), reliability, security, compatibility, performance efficiency, maintainability, flexibility (portability), safety, and interaction capability (usability). Testing is a way of measuring quality, and as such, these quality characteristics can serve to identify testing objectives for a test experiment.

During the experimentation, the actual behaviour of SUT is observed, which is compared with what the SUT should do, i.e., the desired, specified, or required behaviour. This implies that for testing there shall be some idea or information (specification, requirements) of how the SUT shall behave. Sometimes the term ‘oracle’ is used to express the expected behaviour.

Sorts of Testing There are different kinds of testing, depending on the quality characteristics being tested, the degree of accessibility and observability of the system being tested, and the level of system composition being tested; see Fig. 2.1.

Different quality characteristics can be the objective of testing, leading to, e.g., performance testing, usability testing, reliability testing, security testing, and functionality testing.

A traditional distinction in software testing is between black-box and white-box testing. In black-box testing (functional testing or specification-based testing), the SUT is seen as a black box without internal detail, which can only be accessed and observed through its external interfaces. The basis and starting point for black-box testing is the specification, or the requirements, of the system, which shall prescribe what the SUT shall, and shall not do, when observed from the outside. In white-box testing, the internal structure of the SUT, e.g., the structure of the code, is the basis for testing.

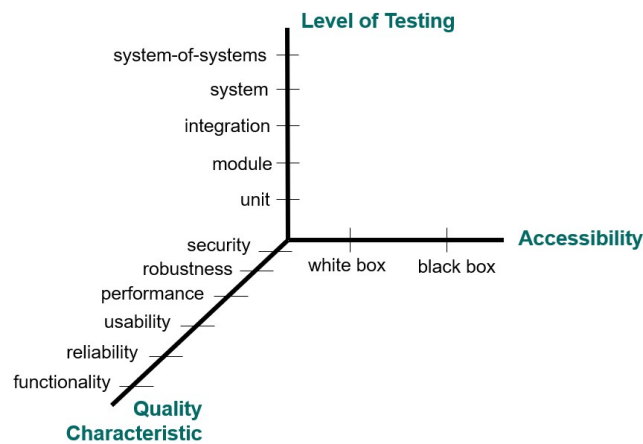


Figure 2.1: Sorts of testing.

Software systems can be tested on different levels of composition or aggregation. On the lowest level, the SUT is called a unit, e.g., a function, method, or procedure of a software program. The next step going up is to test modules or classes having a couple of functions or methods. A number of modules can be combined in a software component leading to component testing. Testing the combination, or integration of modules is also referred to as integration testing. Components can be composed into subsystems which can be tested, and then complete systems, and even systems-of-systems can be tested [5]. The highest level is also referred to as end-to-end testing.

Unit testing is usually simple, cheap, and easy to automate, resulting in a lot of this type of testing to be performed. While going to higher levels, testing usually gets more complex, more expensive, and more difficult to automate. This means that less and less tests are executed on the higher levels. This relation between level and the number of tests is often visualized as the test pyramid; see Fig. 2.2. The test pyramid is also guiding, in the sense that it suggests to do a lot of testing and to detect as many bugs as possible on the lower levels, because that is cheaper and easier than testing on the higher levels. Yet, many properties, such as emerging properties like performance, security, interoperability between components, and end-to-end functionality, can only be tested on the higher levels.

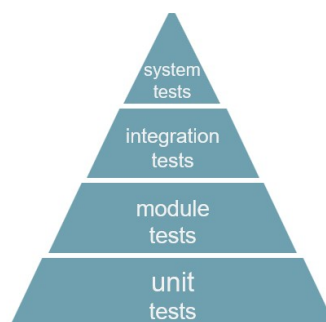


Figure 2.2: Test pyramid.

Another distinction that is often made (not in Fig. 2.1), is that between active and passive testing, or monitoring. In active testing, the tester controls and observes the SUT in an active way by giving stimuli and triggers to the SUT, and observing the responses to these stimuli, as opposed to monitoring, where the tester only passively observes what the system is doing, usually during normal operation, without influencing the SUT. According to the descrip-

tion of ‘testing’ given above, monitoring is not really testing.

In this report, referring to the dimensions described above, we mainly concentrate on active, black-box testing of functionality of the software part of CPS. This by no means implies that other kinds of testing are less important, less interesting, or less challenging. Those are investigated elsewhere, e.g., security testing [23], safety testing [24], performance assessment [25], and monitoring [26]. The choice for the current report makes it possible to focus and achieve a certain depth in discussions, while still being broad enough for plenty of challenges, see Chapter 3, that are relevant for CPS in the high-tech equipment industry.

Quality and Testing Testing is one way of checking software quality. There are other methods to assess and get confidence in the quality of a software product, such as static analysis, code inspection, reviewing, formal verification, model checking, and many more. In static analysis, the software code is (automatically) analysed for static properties such as well-typedness, correct variable declarations, and coding conventions [27]. As opposed to testing, static analysis does not require execution of the SUT itself. Inspection and review consist of thorough human checking of code or other development artifacts. Also with this method, the SUT is not executed. Model checking is a form of formal verification, where a model of the SUT is checked for required properties, supported by tools such as model checkers [28]. An abstract model of the behaviour of the SUT is constructed, and then formal verification aims at showing that a system has specified properties by proving that this model satisfies these properties. Thus, the model is checked and not the real, executing system, which means that any formal verification is only as good as the validity of the model on which it is based. Testing, on the other hand, works on the real, executing SUT, but due to the inherent limitations of testing, such as the limited number of experiments that can be performed in a reasonable time, testing is never complete and can never prove properties: “testing can only show the presence of errors, not their absence” [29]. Model-based testing intends to combine models and testing, by showing through testing that the real, executing SUT behaves in compliance with its model [30, 31, 32, 33]. None of these methods, naturally, is the silver bullet for checking software quality. Only a well-chosen combination of different methods can lead to high-quality products.

Testing, and the methods mentioned above, are product-oriented ways of assessing quality, i.e., the product itself is central in determining quality. An alternative is process-oriented quality assurance: not the product is core, but the process with which the product was designed, developed, implemented, and tested, with the underlying assumption that, if the process is good, then also the product made during this process will be good. Examples of this approach are IS-9001 [34], CMM (Capability Maturity Model) [35], and other SPI (Software Process Improvement) approaches. A third approach to quality, is to put the people doing a specific task central, with the underlying assumption that, if a well-educated, sworn-in, specifically appointed, knowledgeable person performs a particular task, the result will have high quality. Examples are mainly in other disciplines and not in software engineering, e.g., a chartered accountant or a sworn-in notary.

Traditional Testing Challenges In this report, a number of challenges for testing of CPS will be discussed in Chapter 3. The emphasis is on CPS specific challenges, challenges with increasing importance, challenges that are new, or that can be expected in the (near) future. There are also a couple of more ‘traditional’ challenges that are well-known in the testing community, and that have been discussed extensively in the general testing literature [18, 19, 36, 37, 27, 20, 16, 21]. They are mentioned here for completeness without further discussion.

The SUT:

- testing without (clear) specifications or requirements;
- dealing with parallelism, concurrency, and race conditions in software, leading to uncertainty, nondeterminism, and flaky tests.

Sets of tests (test suites):

- what is a 'good' test suite, and how to get or make them;
- how to determine the coverage of a test suite (a-priori coverage) or after test execution (a-posteriori coverage);
- when to stop testing, i.e., when is there sufficient confidence that the SUT is of good quality.

Regression and re-testing:

- test-suite maintenance and maintenance of test automation;
- test selection and prioritization from an existing test suite, in particular, for re-testing of new versions/patches;
- how to speed up re-testing of new versions/patches.

3 Challenges

Testing cyber-physical systems (CPS) is challenging, and testing future generations of CPS might even get more challenging. A number of challenges are discussed in this section; no solutions are proposed. The emphasis is on the software part of CPS, cf. Chapter 2. The order of the challenges does not imply any importance or priority. The classification into different challenges is subjective. There is overlap and there are relations between different challenges, some challenges could be split or combined, and some challenges may reinforce each other or the traditional testing challenges described in Sect. 2.2.

3.1 Multi-Disciplinarity

The first challenge is due to the nature of CPS. CPS have a physical part and a software part that must work together to provide the required functionality. To test such systems completely, knowledge needs to be combined from different disciplines: on the one hand physics, mechanics, optics, mechatronics, electronics, etc., and on the other hand software engineering, data processing, etc. These different disciplines are supported by different theories and modelling formalisms, e.g., continuous systems modelled with PDE (partial differential equations) vs. discrete systems modelled as Turing or state machines. They lead to different views on the system, e.g., a modern car viewed as a mechanical object with computational control units or as a computer on wheels, also called a ‘software-defined vehicle’. These disciplines use different development processes, e.g., waterfall-like processes for the physical parts, vs. agile for software. And they have different testing methods and fault models, e.g., reasoning based on statistical independence of failures for physical systems that wear out, vs. discrete output and transition faults of software, where any fault is always a design-time, coding, or update bug. Lastly, they require different expertises and different people skilled in these disciplines. All this makes testing CPS difficult, demanding, and challenging, even if the focus is on only testing the software part of CPS.

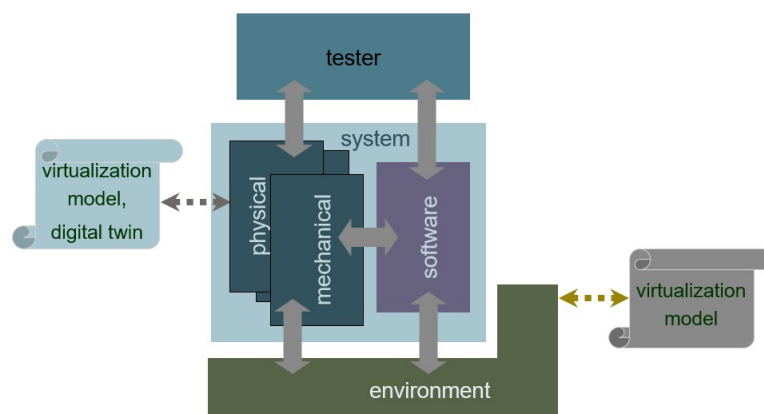


Figure 3.1: Multi-disciplinary cyber-physical systems and their modelling.

Because of these differences and challenges, and also because copies of the physical parts are almost always very expensive and not readily available, the software part is usually tested in isolation as much as possible. In order to do so, the physical part is simulated, that is, the behaviour of the physical part is mimicked in software, referred to as stub, mock, test double,

virtualization, digital twin, or just simulation model. Also the external system environment, e.g., weather conditions, can be modelled and simulated in this way during testing. This implies that not only the real system is tested, cf. the Def. of ‘testing’ in Sect. 2.2, but that also a simulation model can be part of what is being tested.

A simulation model of the physical parts allows to test the software without its physical parts, yet, in a context that behaves as these physical parts. Advantages are that the software part of a CPS can be tested without having the physical part that can be expensive, difficult to obtain, dangerous, polluting, and unwieldy [38]. Moreover, the real physical part can be very slow, e.g., a storm-surge barrier [39], or very fast, whereas in a simulation model, progress of time can be adapted to fit testing purposes. Also triggering specific error scenarios, fault injection, or environment conditions, e.g., weather conditions when testing autonomous vehicles, is much easier in a simulation model than in a real system or environment.

On the other hand, a simulation model is always an abstraction of the real physical part of the system and its environment, and the reaction of the CPS software part to every behaviour that is not precisely simulated is not tested. In addition, developing simulation models takes time, can be expensive, and still requires the combined expertise of software engineering and the physical domain in order to make truthful, precise, and valid models. It might be that different models are required for testing various situations, scenarios, and properties. Model development in itself is a challenging expertise, making the right model for various situations, for different quality characteristics, and on the appropriate level of abstraction. And then, these models have to be verified and tested, too.

3.2 Size and Complexity

The size and complexity of CPS software is ever-growing, e.g., counted by lines of code (LoC), by the number of inputs, outputs, or internal states, by the causality (dependency) between inputs and outputs, or by the number of components or microservices. The consequences of this growth for testing are often underestimated. A larger system leads to an increasing required testing effort, but, unfortunately, this increase is not linear in the size of the system. Worst-case, it is even exponential, in particular, if there are many causalities and dependencies among components and between inputs and outputs, as Fig. 3.3 illustrates. Consequently, in order to keep the same confidence in the system, the number of test cases should also increase more than linearly, which is usually not feasible. The exponential increase leads to the situation that testing cannot keep pace with the development of ever-larger systems, hampering further innovation.

Car <ul style="list-style-type: none"> • 100,000,000 LoC • 40,000 parts • 4,000 manufactured components 	Completely testing ‘+’ for 32-bit Int <ul style="list-style-type: none"> • $2^{32} * 2^{32} = 10^{19}$ test cases • 1 nsec / test = 585 years of testing
Machine with 300 (Boolean) parameters <ul style="list-style-type: none"> • $2^{300} = 10^{90}$ different configurations (cf. #atoms on earth $\approx 10^{50}$) 	

Figure 3.2: Combinatorial explosion leads to excessively large numbers of potential test cases.

A couple of trends aggravate the size and complexity challenge. The first trend is the habit to make almost anything in modern software parameterized. From a testing perspective, a

parameter is yet another input to the system, which multiplies the (theoretical) testing effort with the number of values that the parameter can take. A second trend is to reuse software, leading to (transitively) include many libraries which make software executables unnecessarily large, claiming that if it doesn't help it doesn't hurt. For testing, however, it may hurt. The linked software may contain bugs which are now imported into the software, in particular, for security bugs the attack surface may increase dramatically. Moreover, when the software evolves, this may lead to interoperability problems; see also Sect. 3.4. A third trend specifically for CPS software is that often system functionality is shifted from the physical part to software, sometimes to the level that software has to correct or compensate for physical imperfections. Examples are a simple mechanical valve in a storm-surge barrier that is replaced by sensors, actuators (pumps), and software controlling the pumps based on sensor values, and software in a wafer-lithography system that compensates for physical imperfections of wafers.

Test automation may raise the bar in what can be tested in terms of size and complexity, but it cannot cope with the exponential increase. Moreover, traditional test automation introduces new challenges in the size, management, overview, and maintenance of test suites, in particular, when systems evolve; see also Sect. 3.4 and 3.10.

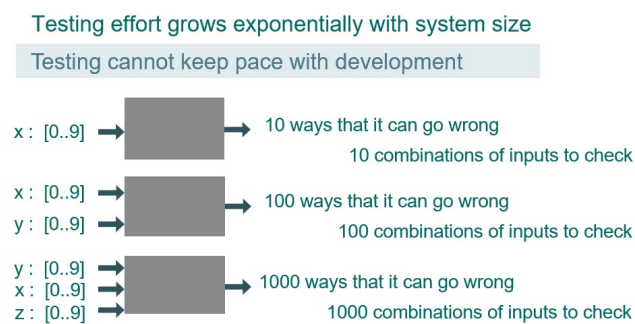


Figure 3.3: Testing effort grows exponentially with system size.

3.3 Variability

High-tech systems come in many variants, customized for different users using various features, options, and configurations. This is often referred to as variability, product line, product family, ‘mass customization’, ‘series-of-one’, or platform-based development. Variability engineering tries to balance between, on the one hand, craftsmanship of building individual and customized products and, on the other hand, mass production with optimal reuse. Different concrete products, i.e., product variants, are built by instantiating and choosing parameters and features in a product line or platform [40].

In product-line engineering, the goal is to instantiate as late as possible, and to perform design, implementation, analysis, and verification as much as possible on the product line or platform level instead of for each possible variant, in order to reduce effort, time, and cost of developing various products. This is also called ‘family-based’ or ‘lifted’ analysis.

Testing, however, involves executing experiments on a concrete, instantiated product. Testing a not-instantiated product line is impossible. So, what looks like an advantage for design and implementation, turns out to be a challenge for testing: ‘lifted testing’ is not an option, and each concrete product should be tested individually. But due to the combinatorial explosion of the number of possible variants, cf. Fig. 3.2, testing all of them is usually not feasible.

The alternative would be to test only those products that are actually delivered, but then for each new customer or delivered product all testing must be repeated.

Consequently, variability adds another dimension to size and complexity, making testing harder. Also here, due to the combinatorial explosion, the testing effort grows exponentially with the number of variation points, features, or configuration parameters, implying that it is impossible to test all relevant behaviours for all instantiated products of a product line. The challenge then is how to cope with the explosion of variants, and to make clever and reasoned selections of variants for testing.

For CPS, variability can be in the physical part and in the software part, with usually many dependencies between them, e.g., particular software controllers or drivers must be installed depending on the configuration of physical components. This also leads to having configurable simulation models for the physical components, cf. Sect. 3.1, which, again, increases complexity, but which may also help testing variability, enabling faster switching between different simulation model variants [38].

3.4 Evolution

Systems evolve over time through updates, upgrades, and replacement of components, in order to fix faults and adapt to changing requirements, user needs, and contexts. Change over time is called evolution, and leads to different versions of a system [3, 4].

Evolution leads to yet another dimension of complexity, in addition and orthogonal to inherent complexity (Sect. 3.2) and variability (Sect. 3.3) [7]. Each variant can have different versions, and a new version can spawn new variants, which all have to be tested. Ever faster, continuous delivery of new versions requires faster testing of updates, changes, and additions, leading to continuous testing (CI/CD – Continuous Integration/Continuous Delivery/Deployment). More and more often, testing is the bottleneck when bringing upgrades and new versions to the market. Thus, evolution makes testing more challenging.

In order to speed up testing of evolving systems, in addition to test automation, change impact analysis might help to decide what to re-test and what not, thus reducing testing effort for upgrades and new versions. The field of systematic change impact analysis for testing is still in its infancy.

3.5 Components

Decomposition is a standard design and development technique for building large and complex systems: such systems are built from sub-systems and components, which, in turn, are decomposed. Components are heterogeneous: some components may be newly built for a particular system, others are legacy, acquired externally from suppliers, open source, outsourced, or off-the-shelf. All these components are very different in terms of quality, available specification and documentation, openness, and the depth of testing they have undergone. Decomposition into components helps in mastering complexity, for outsourcing parts of development, and in reuse of components developed elsewhere. Component-based development involves a multi-level, hierarchical approach, starting small with functions, methods, often referred to as units, combining these into larger modules, then into components and larger components, sub-systems, eventually into a system, and perhaps even into systems-of-systems. In software testing, this hierarchy is often represented in the form of a *test pyramid*; see Fig. 2.2.

Component-based development requires a clear testing and integration strategy, which comes with a number of challenges:

- An important question is where to concentrate test effort, and to guarantee that correctly tested components integrate into correctly functioning systems. Testing can be on the lower levels in the test pyramid, where testing is usually cheap, fast, and automated, but where not the whole system is tested, and in particular, interactions between components are not tested. Testing can also be on the higher levels where testing is more expensive and slow, often manual, and building a test environment is cumbersome, but more interactions and functionalities can be tested end-to-end. The general opinion is that every fault should be detected on the lowest level possible. It is not always easy to find out which level that is, and to select which tests still have to be performed on the higher levels, if lower-level tests have been done. Optimization of testing over the levels of the test pyramid is a challenge.
- If components come from elsewhere (reuse, outsourced, open source, legacy, etc.), then an important question is to what extent these shall be (re)tested, taking into account that reuse of external components provides additional functionality but probably also additional bugs. Strategies for acceptance and testing of external software range from complete trust of the supplier (no testing by the acquirer at all), conformance statements (no testing but the supplier declares correctness and takes responsibility), evidence of testing by the supplier (component-acceptance testing), light-weight retesting by the acquirer, until complete re-testing of components by the acquirer, which, of course, can be very costly.
- An additional challenge pops up, in particular for CPS, if also physical components have to be integrated, or if simulation models of these physical components are used in testing. In case this concerns the real, physical component, testers are confronted with the challenges described in Section 3.1. When a simulation model is used, the question is to what extent this model has been, or shall be tested, and to what extent it can be trusted to show valid behaviour.
- Also components come in different variants and configurations; cf. Sect. 3.3, with usually many dependencies between components, i.e., some component variant can only work with a particular variant of another component. More and more often, system failures are not caused by coding errors, but by wrong combinations of component configurations and parameters. The number and variation of these configuration parameters is ever increasing, leading to yet another combinatorial explosion that complicates testing.
- Analogous to the combination of compositionality and variability, also the combination of compositionality and evolution creates yet another source of combinatorial complexity [41]. Systems evolve, e.g., by replacing components, but also components themselves evolve. The new component itself must be tested for correctness, but also for correct interoperability with its environment, i.e., the other components in the system; cf. the infamous Ariane V failure [42]. Like for variants, not every version of one component can interoperate correctly with every version of other components, so, the integration shall be tested. Since testing all combinations is not feasible, again, a selection has to be made, and arguments for such a selection must be reasoned.

A particular instance of the version problem is the inclusion of external or system libraries in software code. Usually, these libraries (components) are not tested when imported. Both the importing software and the imported libraries evolve, which at some point may lead to interoperability issues. Moreover, there is the well-known dilemma

of software developers to include a fixed library version or the latest. In the first case, you do not get (security) bug fixes, but in the second, the library may evolve leading to interoperability issues even if you do not change your own software.

A currently popular architecture for decomposition of monolithic systems into subsystems, is the *microservices architecture*. This will be discussed separately in Sect. 3.13.

3.6 Connectivity

Yet another layer of testing challenges is formed by systems not operating in isolation, but being connected to other systems, forming systems-of-systems (SoS), which provide functionalities that the individual systems cannot provide [43, 5]. The difference between components (or sub-subsystems) composing a system and systems composing a system-of-systems is governance, ownership, and independence. Whereas a single system is designed, operated, and controlled by a single party, there is no single party that operates, controls, and is responsible for an SoS [44]. Connected systems and systems-of-systems pose new testing challenges:

- Connectivity, naturally, adds to size and complexity, usually combinatorially, making testing more challenging, as discussed in Section 3.2.
- For testing systems-of-systems, that is, systems that are connected to many other systems, and that depend on those other systems for providing functionality, global specifications of behaviour are usually missing.
- Testing an SoS means that the tester has no ownership and control over some systems in the SoS, which leads to uncertainty and complicates testing. Moreover, during the testing phase of the own system, these other systems are usually not available, necessitating the development of simulation models for these other systems, which leads to extra cost, effort, and the question of validity of these models.
- No complete ownership and control over an SoS implies that the other systems can be updated, adapted, or removed from the SoS, or that a new system can join the SoS without notice. The SoS configuration is dynamic and can change during operation, and this configuration is not known during the design and testing phases. Consequently, complete testing during development is not possible. Instead, this form of dynamics leads to the necessity to extend testing to the operational phase, either as active testing or as monitoring (passive testing); see Sect. 3.11.
- Connecting a system to another system may make it more difficult to determine what exactly the SUT (System-Under-Test) is and what its test-interfaces are. The boundaries between a system and its environment become blurred.

3.7 Quality Characteristics

For software systems, the current, commonly used quality model is ISO/IEC 25010-2023 [22], with 9 main software quality characteristics; see Sect. 2.2. A quality model is not fixed and may change over time, with adaptations made to the quality characteristics. ISO/IEC 25010:2023 was preceded by ISO/IEC 9126:1991, ISO/IEC 9126:2001, and ISO/IEC 25010:2011, where “security” and “compatibility” were added as main characteristics. In ISO/IEC 25010:2023, “safety” (and some sub-characteristics) was added compared with ISO/IEC 25010:2011.

Classical testing, and many classic testing books, mainly deal with testing of the quality characteristic “functionality”. Over time, other quality characteristics increased in importance, e.g., security, safety, usability, and, consequently, testing for these quality characteristics also became more important, also leading to new testing techniques specific for these quality characteristics, e.g., fuzzing for security testing [45, 46, 47]. From a testing perspective, a challenge is that many of these other quality characteristics are non-compositional, sometimes referred to as *emerging properties*. Whereas testing of components for functionality contributes to confidence in the correctness of the system built from these components, this does not always hold for emerging properties. This means that testing components for emerging properties may not help in achieving these properties on system level. In other words, the test pyramid, see Fig. 2.2, loses its value, implying that emerging properties can only be tested by expensive and labor-intensive tests on the highest level, sometimes even on systems-of-systems level. An example is safety for cars, which is considered on SoS level, i.e., not testing individual cars but sets of cars.

In the future, new quality characteristics, i.e., new kinds of properties for ICT systems, may pop up. Candidates are *sustainability*, *fairness* (for AI), and *ethics* [48]. Consequently, new testing methods and techniques may have to be developed for these new quality characteristics, such as sustainability testing and fairness testing.

3.8 Environment

When testing, there is a system under test (SUT), but also the environment of the SUT. Traditionally, the emphasis is on studying and engineering the system: the system is specified, designed, implemented, tested, and deployed, and the reasoning is from the point of view of the system, i.e., the system is described and specified in terms of accepted inputs, provided outputs, and perhaps some internal states of the system itself. An example is a traditional calculator, which is completely specified by the relation between numbers representing inputs and outputs representing the result of calculations on these numbers.

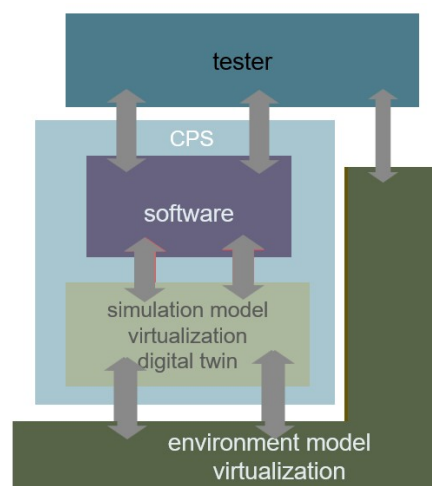


Figure 3.4: System Environment.

For ever more modern systems, there is an increasing role for the environment. Requirements on the system are expressed depending on the environment, e.g., “an autonomous vehicle (AV) shall be safe and shall not collide with other vehicles.” Satisfaction of such requirements depends on the behaviour of the environment, i.e., the other cars, but also the weather may influence the probability of a collision [49]. Generally, required system behaviour

is expressed with respect to some possible behaviour of the environment. Safety is a quality characteristic that is typically specified in terms of a system behaving in its environment, but also security threats can be attributed to environmental actors (a threat or attacker model). Consequently, when testing for such requirements, the behaviour of the environment, e.g., the behaviours of the other cars, must be taken into account [50].

One might think of the environment as just another input, e.g., in an AV it manifests as input data to sensors that is then processed by the system. A test case could be constructed by providing inputs to these sensors and observing the reaction of the system. With such a view, however, it would be difficult to take into account the relations and dependencies that exist between different inputs in a realistic environment, both in space – different sensors making observations of the same object – and in time – a sensor observing the same object on different moments in time. Not considering these relations and dependencies would lead to many, or almost all test cases not reflecting realistic situations, actually testing would be more like fuzzing [46]. It would be difficult to make a test case that tests whether an AV correctly recognizes another car. Taking these relations and dependencies into account would require specifying them, e.g., as constraints over the inputs. But then, most likely the easiest way to do that, is to take the environment itself, and let realistic test inputs be generated from environment data, or from a model or simulation of it. Moreover, many requirements are specified with respect to an environment. A car shall not recognize all possible combinations of inputs on its sensors, but only those that matter in a realistic environment. Also requirements about safety, security, reliability, or availability are usually specified with respect to some environment, e.g., a safety case for safety testing, or an operational profile for reliability testing. No car is completely safe in all possible environments and, consequently, testing in all possible environments is not practicable and not useful.

Environments are very heterogeneous and diverse, showing many different behaviours, they can be friendly (co-operative) or adversarial (hostile), they concern different (quality) characteristics, they can consist of software, physical processes, or humans, they can be predictable, less predictable and uncertain, or completely unpredictable and chaotic, e.g., the weather when testing an AV. Moreover, environments can change in directions not foreseen when designing the system, and system and environment may mutually influence each other, sometimes in ways not anticipated in advance. Consequently, many different environments with a myriad of varying behaviours must be taken into account, thus adding yet another dimension of high complexity to testing.

The physical part of a CPS is part of the friendly environment of the CPS software. In contrast, the weather, e.g., a thunderstorm, is a potentially adversarial environment, while another car can be both friendly and adversarial for an AV, e.g., it is friendly by driving on the right side of the road, but yet, it may still make movements causing it to collide.

Testing a system in all its possible environments and for all possible current and future environment behaviours is impossible, and making selections over environments for testing is even more challenging than selecting test inputs for the SUT. Moreover, it is difficult to elicit specific environmental behaviour when testing in real environments, so, like for the physical part of a CPS (Sect. 3.1), the environment must be modelled and simulated for testing [24]. On the one hand, this makes it possible to simulate all kinds of environment behaviour that occurs rarely in reality, yet, on the other hand, any behaviour that is not explicitly modelled in the environment model, will never be tested at all (the infamous ‘black swan’) [51]. Having valid and trustworthy environment models, obtained by explicit modelling or by derivation and learning from environment data, is crucial for testing systems that can operate correctly in diverse, uncertain, and potentially hostile environments.

3.9 Climbing the Pyramid

Unit testing, i.e., testing the code of functions, methods, and classes, constitutes the lowest level of the test pyramid; see Fig. 2.2. This kind of testing is usually performed by the programmers themselves and not by testers, it is supported by various tools, e.g., JUnit, and it is often integrated in CI/CD (continuous integration/continuous delivery) pipelines. Many projects and companies perform unit testing, also since it is relatively easy and cheap, yet, it can detect quite some coding errors.

Many bugs, however, only occur when many units interact, e.g., integration, communication, synchronization, and concurrency bugs. Also testing of user functionality, conformance to requirements, and end-to-end testing must be done on higher levels, such as integration, subsystem, or system level, or even system-of-systems level; see Sect. 3.6.

Testing on these higher levels is more challenging, takes more effort, and is more expensive. Higher-level testing is often black-box and cannot use the information in the code. Some kind of specification or requirements are needed to know what to test (which are sometimes missing, cf. Sect. 3.10). Higher-level tests are more involved and more difficult to automate, usually requiring some kind of domain-specific test tooling to access and observe the test interfaces, instead of simply calling functions and methods as on unit level.

Moreover, not only the developed system code may contain bugs, but also ‘meta-code’, such as build scripts, configuration and parameter settings, orchestration tools, containerization, deployment tools, run scripts, etc., may contain bugs that prevent the system from functioning correctly. More and more bugs detected at system level, also after deployment, concern bugs in such ‘meta-code’.

Another challenge of going-up in the pyramid is abstraction. Naturally, higher-level tests lead to abstraction from lower level details. An abstraction gap may occur between abstract system-level requirements and the concrete test cases that test for these requirements. An example is a printer printing a pdf-file. An abstract action may be “print pdf-file”, whereas the concrete actions on SUT level would involve steps like creating a valid pdf-file, warming-up the printer, inserting the pdf-file in the print queue, taking a sheet of paper, printing the sheet with the first page, taking the next sheet, Another, extreme example is the requirement given in Sect. 3.8 stating the required safety of a car, which is a very abstract requirement, that has no direct relationship to steps to be performed on the SUT to concretely test for this property. In addition, abstraction from details, e.g., the details of an internal decision in the SUT which we do not know, or do not wish to know, may lead to uncertainty, non-determinism, or probabilities in tests. This means that the expected output of the SUT cannot be uniquely predicted anymore.

Going-up can go even further, beyond developing and testing single systems as products. There is a trend towards ‘servitization’: offering services or ‘solutions’ instead of systems, software-as-a-service (SAAS), and cloud solutions, aimed at minimizing total cost of ownership. A question then is what the important quality characteristics of such services and X-as-a-service solutions are, and how to test these.

Altogether, these challenges lead to the preference for low level, cheap, easy unit tests, yet, also high-level tests are necessary. A shift-down towards unit test could reduce cost, but not everything can be tested on that level. An important question is what to test on which level of the test pyramid of Fig. 2.2, optimizing over cost, required coverage, and quality.

3.10 Agile and Test Automation

If agile software development is applied, usually at the lower levels of the test pyramid, many of the artifacts and deliverables that are typically used in the traditional way of developing software, are absent or are changed as the project evolves. The software developers are expected to code and test in quick succession, where tests serve as flexible, partial specifications of the behaviour of the code. This test-fast-and-often paradigm requires a high level of test automation, usually within CI/CD pipelines.

As a consequence, large repositories of scripted tests are created. Maintaining these test repositories is difficult, since there are no objective means for establishing whether tests are redundant, overlapping, or even meaningful. The sheer size of the test repositories poses a test execution problem, causing the testing process to take an excessive amount of time and requiring a lot of resources. Moreover, these test repositories tend to grow by adding tests, whereas removing tests turns out to be much more difficult, in particular, if added by someone else, probably not in the project anymore, and without clear documentation what the test really tests.

These test repositories of fixed, scripted tests suffer from the so-called ‘pesticide paradox’: each time executing exactly the same tests will not discover any new faults, giving a probably false feeling of quality of the product [27, 21]. Variation in tests is necessary, but difficult to achieve. If, on the other hand, the system is modified, these tests are often used as implicit system specification, i.e., as a safety net catching regression faults. This leads to too much reliance on these tests, since they are more often than not incomplete as an implicit specification.

While software developers may contribute quite a number of tests during development, those tests often remain at unit level. Since in an agile process, it is often not precisely known what the system should do when coding starts, there is not always a prior agreed upon (functional) specification that the system should meet. Thus, test engineers do not have a solid basis to start designing their tests, in particular, for tests at higher levels beyond unit testing.

3.11 Runtime Testing

A traditional software-system life-cycle model consists of a number of linearly ordered, sequential phases, such as inception (the start: what are we going to do), development (requirements analysis, design, implementation, testing), release (“throwing it over the wall” and developers are done), deployment (making it work in some production environment), operation (continuous use of the system, including maintenance: fixing bugs, improvements to meet new requirements, adapting to changing environments), and decommissioning (transferring to a new, other system). An example of such a model is the classic waterfall model. Though not necessarily executed sequentially, such a model does represent a conceptual ordering. Testing is a phase in such a life-cycle model.

Modern software-development life-cycles are not linear and sequential, but more cyclic and concurrent, where phases are intermingled [6]. For example, in *DevOps*, the work of development ‘dev’ and operations ‘ops’ are integrated, which is often represented in the ‘DevOps pretzel’; see Fig. 3.5. The system evolves constantly, while continuously new versions are developed, using data that is obtained from monitoring systems during operation, and applying agile processes; cf. Sect. 3.10. At any moment, there is a version in any of the phases, making the phases concurrent and continuous: CI/CD, i.e., continuous integration and continuous delivery/deployment. The goal is to have planned modifications quickly available in operation,

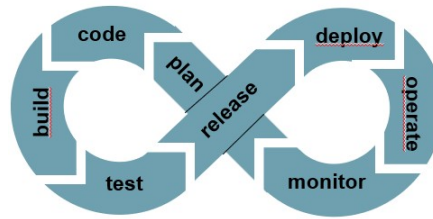


Figure 3.5: DevOps Chain.

without sacrificing quality. A modification can be a fix, an improvement, or an adaptation, it shall be small to speed up its processing through the phases, and the activities in the different phases shall be automated as much as possible.

Consequently, also testing in DevOps shall occur continuously, test suites shall be constantly available, they shall evolve with the evolving system, and testing shall be fully automated. More and more often, testing becomes a bottleneck in keeping up the speed of continuously changing and updating systems. Maintenance and adaptation of test suites following the planned and implemented system changes and updates, i.e., test change impact analysis, is a growing challenge.

Future systems may even go a step further. Whereas in DevOps, testing is still a development activity, see Fig. 3.5, such pre-deployment testing may turn out to be insufficient for next generation systems. Such systems, or systems-of-systems, will not only deal with planned changes for repair, improvement, or adaptation, but they will also have to deal with unplanned, dynamic changes, i.e., changes during operation. Examples are changes in the environment, in third party components, or in used services, dynamically added, removed, or updated components or services, dynamic reconfiguration, and dynamically changing deployments, i.e., runtime mapping of software executables to different computing devices, usually virtual ones.

This dynamic evolution (runtime evolution) leads to systems that behave autonomously, which means that at runtime, i.e., during operation, they will change their behaviour, adapting to the changes with which they are dynamically confronted. Complete pre-release testing of such systems is an illusion, which means that a way of runtime testing and quality control is required, so that testing can continue after release. Currently, after-release testing is often restricted to monitoring (passive testing). The runtime testing challenge is how to extend testing after release to dynamic and active testing, in order to enhance the correctness, resilience, robustness, reliability, security, and dependability of systems during runtime. The development of such runtime tests, however, is difficult, in particular, they should by no means interfere with the normal operation of the system, and even after a failing test the whole system (of systems) must continue some kind of safe and normal operation.

3.12 AI and Testing

Although AI has been around for decades, the latest developments in AI, in particular developments around LLMs (Large Language Models), make that it is currently among the most disruptive new developments. Also in the area of CPS, AI is becoming increasingly used, in many different ways. It is used, among others, for system development, requirement analysis, software development, and testing, and it can be embedded in the system under development itself. AI is rapidly evolving, the field is still immature, and a lot of things are unclear, posing a lot of new challenges when testing systems. AI brings new testing challenges, but

also new opportunities for testers. We discuss testing of AI-generated code, testing with AI, and testing of AI.

Testing AI-Generated Code AI can be used for programming. It lets programmers (co-)generate code via tools such as GitHub Copilot, ChatGPT Codex, and Claude. While such code may compile just fine, it might be inadequate at solving the problem it is meant to solve. The code is generated starting with a prompt, which then serves as a kind of implicit specification for the code. Testing a program generated with AI is not that much different from testing a traditional program, only that it may be unclear what the basis for testing should be and what is then actually tested. The generated code can be tested (white box testing) but with respect to what, i.e., what is the oracle. The code can be tested with respect to the prompt specification, but this may raise questions about the correctness of the prompt and it tests more the AI code generation process than the resulting code. The generated code can be tested with respect to an independent specification or requirements, which must be available, and this poses questions about the relation between this specification and the used prompt. AI could also be used to (reverse) engineer a specification from the code. In general, a question is whether the automatic or supported generation of code, or of any other engineering artifact, with AI or with some other tool, will change the relation between these artifacts. This includes how this generation will influence the testing strategy, which artifacts will be tested with respect to what, and the kind of properties that will be tested.

Testing with AI It is hoped, and expected, that AI will contribute to improvements in testing, and to solving some of the testing challenges, e.g., by better analysis of requirements, faster generation test cases, and more precise analysis of test outcomes. It is yet a challenge to figure out for which activities and how best to deploy AI to increase the effectiveness and efficiency of testing. Currently, there is a lot of experimentation, mainly for lower level testing like unit testing [52, 53], and with positive as well as negative results. Additional research has to be carried out to investigate whether AI can indeed improve the efficiency and effectiveness of testing. Possibly interesting questions are whether or how AI-generated code can be tested with AI-generated tests, without getting into an incestuous situation, and whether faults detected with AI-generated tests could also be repaired with AI.

Testing of AI Systems that contain AI, e.g., decisions or outputs of the system are determined by ML (Machine Learning) models learned from data, are fundamentally different from traditional algorithmic systems. Neural networks, for example, pose a challenge when testing them since they are based on training the net using large amounts of data. For such networks, it is often impossible to describe the desired behaviour, and it is currently also not clear what types of properties should be tested. One may think of robustness of the net, but also determinism and quality of the net, in particular, as it evolves over time. While it is clear (though perhaps difficult) what to do with test failures in classical software, the same cannot be said of software whose behaviour is largely determined by the neural network. Moreover, there are different artifacts that can be tested in case of a neural net: shall the machine-learning (ML) algorithm that generated the model be tested, or the training data that is input data to ML-algorithm, or the running system processing user data? Traditional ML-testing splits the training data so that part of it is used for training and another part is used for testing. The proverbial black swan will probably not be detected by such a test set that is analogous to the data set that was used for training the system.

A next step in testing of AI-based systems is when the AI-based system can continue learning during operation, adapting its model and its behaviour to new situations; see dynamic testing in Sect. 3.11. Systems will increasingly incorporate AI, but testing such systems is still in its infancy.

3.13 Microservices

Microservices, or microservice architecture (MSA), are an architectural approach in which a software system is decomposed into a collection of smaller subsystems, called services, that are loosely coupled and work together to provide the functionality of the system [54, 55]. They can be seen as a further evolution of component-based development; see Sect. 3.5. The services are small and independent programs, that are separately deployed, independently running, and that communicate with each other via messages using standard, lightweight protocols and well-defined API-interfaces. A service implements a single (business) functionality, it has its own data, resources, and domain logic, and it does not assume any shared resources or a specific host machine. This means that there is a flexible mapping of services to computing hardware, often using virtual machines or containerization. Moving to the cloud often comes with the need to go from monolithic architectures to microservices.

Each service is an independent component, that is developed and owned by a single team, using its own implementation technology and programming languages, leading to technical diversity in the collection of services. The stable points are the technology-agnostic communication protocols (often http-based) and defined APIs for communication between services. A microservices system (MSS – a system built as a composition of microservices) is a distributed system.

Being independent implies that microservices can be independently developed, built, released, deployed, executed, decentralized, updated, and stopped, thus reducing code interdependency, while improving modularity, flexibility, scalability, and evolvability, as long as they comply with their APIs. A change to a small part of the system only requires rebuilding and redeploying the microservices involved. Lack of performance of a particular microservice can often be solved within that microservice by scaling computational resources or running multiple instances of that service. MSA pairs well with agile development processes, continuous integration and delivery (CI/CD), and automation of the whole release, deployment, and execution process, thus leading to faster development cycles. Though less common than in the domain of traditional enterprise applications, MSA is increasingly used in the CPS domain [56].

While there are many benefits on the development, deployment, maintenance, evolution, and management side, the use of microservices does come with some challenges on the testing side. Actually, almost all of the previously discussed challenges cf. Sect. 3.2 – 3.11, appear when testing microservices.

Size An MSS can be composed of hundreds or even thousands of microservices [57]. Such a system is huge in size, making it difficult to test it as a whole (end-to-end testing), or to choose and identify what exactly the product or SUT is that shall be tested. When testing a composition or cluster of microservices it may be difficult to identify what belongs to the SUT and what is part of its environment, even more so when some of the services are external. If not the whole MSS is tested but only a subset of cluster of services, then a testing approach is needed that can cope with such clusters and involves compositional testing, stubs, mocks, test doubles, and partial system specifications. Having a large MSS also poses challenges, after a bug has been detected, on bug analysis, diagnosis, root-cause analysis, and reproduction of bugs in a concurrent system.

Complexity Individual services, each of which concentrates on one or a limited set of tasks following the adage “do one thing and do it well”, have usually less complexity than a monolithic system. The complexity, however, hits back in the interaction and communication be-

tween microservices. An MSS is a distributed, concurrent system, with all the complexity of synchronization, communication, concurrency, network latency, emergent behaviour, non-determinism, and race conditions occurring in such systems, which make testing of distributed systems a complex and challenging task. In addition, inter-service communication in microservices utilizes complex data objects due to the fact that services have their own data and cannot resort to shared data resources, which leads to extra complexity in test-data generation and building test harnesses.

Variability Microservices are deployed in different environments, cloud contexts, and orchestration frameworks, which means that they must be configurable for these diverse environments. Such configuration and parameter settings are also prone to errors. Indeed, ever more failures in large systems are due to configuration and parameter bugs as opposed to coding errors.

Like for testing variability in general, see Sect. 3.3, there is the configuration selection problem for testing MSS, i.e., the dilemma to either test for a particular configuration, so that testing must be repeated for each change of configuration or environment, or to test for all possible configurations up front, which is usually an infeasible task, the more since also the environment is configurable.

Evolution Microservices systems have fast development cycles which make them constantly evolve. Evolution does not occur by replacing the whole system, but by replacing some individual microservice or clusters of microservices. The fast development cycles lead to the need for fast, automated testing. Replacement of microservices leads to the need to test at runtime, see Sect. 3.11, and to test for different versions and version compatibility, since different versions of a service may run in parallel. If in addition to evolving behaviour of microservices with the same API, also API definitions themselves change, testing naturally gets more complex.

Components Microservices act as components in larger systems, the difference being that they are not statically included but dynamically, i.e., at runtime. Examples are auto-scaling, i.e., dynamically creating more or less instances of a service to manage performance, and adding services when they are dynamically becoming available and detected through service discovery. This means that, in addition to all the challenges of Sect. 3.5, testing has to deal with dynamically added, updated, and changing components. If these are external components, then version and configuration information may not be known in advance, so that this information can only be determined dynamically, e.g., through discovery, during testing, introducing uncertainty (nondeterminism) in the testing process.

Connectivity An MSS is itself a distributed system, i.e., a composition of connected communicating services. A second layer of connectivity is constituted by all additional tools, utilities, and platform infrastructure that make microservices run, such as virtualization, container technology, orchestration, load balancers, auto-scalers, cloud infrastructure, configuration settings, service registries, message brokers, and monitoring services. All this additional software influences the quality, and thus the testing of the MSS. Moreover, this additional software also evolves, has versions, and diverse configurations complicating the testing process. Thirdly, microservices often form systems-of-systems with external services, using other services offered “as-a-service”, which challenge the testing process.

Quality Characteristics All quality characteristics of ISO/IEC 25010-2023 apply to microservices (Sect. 3.7). Since microservices communicate with each other via standardized means

of communication, e.g., http, and since they are often run on external platforms (cloud), there is currently a lot of attention for security in microservices systems.

Environment Analogous to connectivity, see above, the environment can be considered at different levels. The environment of one microservice is another microservice, the environment of a set of microservices is the additional software for orchestration, infrastructure, etc., that makes an MSS run, the environment of a running MSS is the external services that are used and the external environment to which the MSS finally offers services. Apart from the layering in the environment there are no additional environmental challenges specific for microservices.

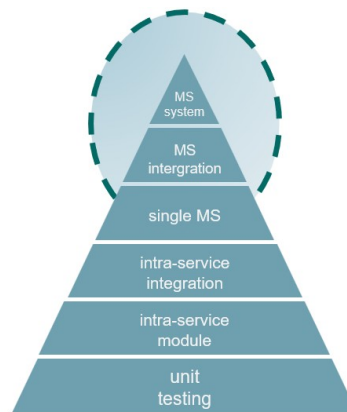


Figure 3.6: Test pyramid for Microservices.

Climbing the Pyramid A single microservice corresponds to a single program, and thus can be tested accordingly using traditional testing approaches. As argued above, the complexity of a microservices system is not in the individual microservices but in the interaction between the microservices. Consequently, there are layers in the test pyramid above testing a single program/microservice. These layers express testing the integration of microservices and testing the final microservices system, as depicted in Fig. 3.6. Also the interaction of the developed MSS with the orchestration and infrastructure utilities has to be tested on these levels. Since on these levels the peculiarities of the MSS as a distributed, concurrent system, with all the complexity of synchronization, communication, concurrency, network latency, emergent behaviour, nondeterminism, and race conditions, has to be tested, it is doubtful whether the pyramid form for this testing can be maintained, or that MSS-testing will turn it into a top-heavy one, expressing that more tests are needed on the higher levels.

Runtime Testing An MSS is a dynamic system, in the sense that microservices can be added, updated, changed, and removed at runtime. Consequently, the reasoning of Sect. 3.11 applies. An MSS cannot be completely tested at development and pre-deployment time. Runtime monitoring and testing will be necessary. Currently, the emphasis is on passive runtime monitoring of MSS, but monitors can only witness issues once they appear. In the future also active runtime testing might be necessary, by adding special test services extending the passive monitors, in particular, if external services are used for which there is less evidence of their quality. This may include API testing and contract-based testing [58].

4 Research Directions

Based on the analysis of testing challenges in Chapter 3, a number of research themes are identified. There is no explicit mapping between the identified challenges and research directions: sometimes this relation is obvious, e.g., when discussing AI and testing in Sect. 4.5, while other research directions are more comprehensive and apply to several challenges, e.g., testability in Sect. 4.1 and modelling in Sect. 4.2. An overall selection criterion for research directions is the perceived importance for TNO-ESI and its partners, but a systematic study of current testing problems and shortcomings in industry was not performed; see also [6, 59], or [8] for a broader context.

4.1 Testability

There are many challenges when testing CPS, and, all too often, testing comes as an afterthought, discovering too late that the developed system is difficult to test. In order to test future generations of CPS, testing must already be considered during the architecture and design phase of the system: how can the system architecture and its design already take into account that at some time the system must be tested?

Testability is the degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component, and tests can be performed to determine whether those criteria have been met [22], i.e., the ease with which a system can be tested. It is important that early in the architecture and development phase, testability is explicitly considered as an important quality characteristic, and that testers are stakeholders in the architectural process, which is currently not often the case [2]. Considering testability in the architecture phase may lead to alternative design choices, or to not include particular features that are deemed untestable, or that may lead to a test effort explosion (e.g., reducing ‘feature fetishism’).

Research on testability shall consider questions like:

- How to measure and express testability of software [60, 61], systems, and their architectures?
- Which system attributes and architectural patterns increase or decrease testability?
- How to include testability in a systematic manner in the architecture and development process?
- How to give feedback and exercise back-pressure towards the architecture and design based on testability considerations?

4.2 Modelling

Increasing size and complexity leads to the need to consider testing at a higher level of abstraction, abstracting from details, and reasoning about the SUT, its behaviour, and its testing using higher level concepts. This naturally leads to using models: “any description of the

system that is not the thing-in-itself (das Ding an sich, according to Kant)” [62]. Though traditional software testing methods are mainly aimed at code-level testing, use of modelling in the testing process is currently widespread. Models can be used for model-based testing; simulation models as stub, mock, test double, virtualization, or digital twin, modelling software as well as physical components, see Sect. 3.1; modelling and simulating the environment; and modelling users, usage, or usage scenarios.

Also during design, development, and implementation, i.e., in the left-hand side of the ‘V-model’, models can be used. This is often referred to as model-based systems engineering (MBSE) [63] or model-based software engineering, comprising activities such as requirements modelling, models to represent, analyse, and document design, visualization, model checking, conceptual analysis, implementation blue-print, and code generation.

Doing many modelling activities may lead to a situation where many different models are used for the same system, with different properties, for various purposes, and used in different activities and phases of system development. Some models are descriptive, e.g., an environment model describes how the environment may behave, whereas others are prescriptive, e.g., a model for model-based testing serves as a specification, i.e., it prescribes how the system shall behave. Some models are merely pictures and serve to support understanding of the system, whereas others are machine processable aiming at automation and generating artifacts from the model such as code or test cases; some models serve to illustrate possible behaviour, whereas others are intended to be complete over all behaviours. Some models have a rigorous, well-defined formal (mathematical) semantics, whereas others have a more informal semantics or rely on intuition.

So, there is a growing use of models in the design, development, and testing of CPS, but it is unclear how all these models relate, whether they can be transformed into each other, whether they can be used for other purposes than they were made for, and whether they are complementary or contradictory for the same system. For example, an often used language for MBSE in CPS is currently SysML(V2) [64, 63], and SysML models help in the design and analysis of systems, but for (model-based) testing they do not seem very suitable [65, 66]. This quagmire of models may hamper further progress in model-based engineering. Practitioners will not be inclined to make models, if they will have to make a dozen different ones. What is needed, is a common view and integral approach to modelling, with a clear understanding of which kind of models shall be used when, for which activities, and how different models relate. Research on such an integral approach to modelling in development and testing may consider question such as:

- Which kind of models exist, what is their purpose, how are they related, and how do they differ?
- Which models can be used for which activities? For example, for state-based model-based testing there are, among others, ComMA [67, 33], the BPMN4S Mátala approach [32, 59], TorXakis [68], and the Axini Modeling Platform [69].
- What are the properties of different kind of models in design, code generation, testing, conceptualization, etc.?
- How to make (good) models, considering that making models in itself is a challenging expertise, in particular, choosing the appropriate level of abstraction?
- How and where can a digital twin be used as model in CPS testing?

- What is the relation between a model and reality (model validation), what abstraction gaps exist, and where does abstraction introduce uncertainty (nondeterminism)?
- What will be the role of AI in modelling, can AI support the construction, modification, maintenance, and use of models, and will the use of AI in system development replace modelling or will modelling and AI be complementary [70, 71]?

4.3 Compositionality

Testing a CPS is challenging and sometimes problematic considering its size and complexity. The standard engineering way to deal with size and complexity is trying to break the problem down into smaller ones that can be solved, that is, decomposition. There are various ways of decomposing the testing problem for CPS:

- decomposition according to quality characteristics, i.e., testing functionality, performance, usability, etc. separately, which is a common approach in testing;
- decomposition according to the test pyramid, see Fig. 2.2, i.e., starting with unit testing, and then continuing with integration tests, component tests, system tests, etc. As discussed in Sect. 3.5, there are some challenges related to testing according to the test pyramid, such as optimization of testing over different layers, and the question under which conditions correctly tested components integrate into a correctly functioning system.
- decomposition of variability into features, where each feature specifies a slice of user-visible behaviour [72, 73], which is then separately tested. Analogous to components, the question is under which conditions correctly tested features make up a correctly functioning system.
- decomposition of evolution into its replaceable and updatable units. Analogous to components and features, the question is under which conditions a correctly tested replacing unit results in a correctly functioning system, and what the additional impact of the replacement on the correctness is.
- decomposition of (existing) test suites, so that they become more manageable.

Decomposition in testing (compositional testing) is often used in an informal and intuitive setting, where the underlying, theoretical justification is missing. Decomposition for variability and components as building blocks has been investigated in the MasCot TiCToC project [74, 75, 76]. More elaborate investigations of decomposition in testing may lead to new possibilities and more systematic analysis to deal with the combinatorial explosion which is the result of combining components. Such new approaches can go further than the classical approaches of combinatorial testing and pairwise testing, for example, by

- decomposing into orthogonal components, e.g., features that provably do not influence each other;
- performing mutual impact analysis, and restricting testing to the impact that one component can have on the other, instead of combinatorially combining all possibilities;
- using alternative methods to check interoperability, i.e., the conditions under which correctly tested components integrate into a correctly functioning system.

4.4 Runtime Testing

In many current and future systems, e.g., autonomous and self-adaptive systems, complete pre-release testing is an illusion. Some form of dynamic or runtime testing and quality control is necessary; see Sect. 3.11. Dynamic, runtime testing comes with many questions, such as:

- What is exactly the SUT that is dynamically tested?
- How to test while not influencing or obstructing normal operation?
- How to develop or generate tests for dynamic testing? If the system evolves continuously and dynamically, then also the tests have to evolve continuously and dynamically, which means that a collection of tests cannot be prepared in advance.
- How to combine, integrate, and complement different ‘self’-quality assurance techniques, such as self-healing, self-protection, self-recovery, self-test, self-monitoring, and self-reconfiguration?
- What is, or can be, the role of AI in such systems and their testing; see also Sect. 3.12?
- How to deal with, and reason about uncertainty and nondeterminism, since at development and design time, there is a lot unknown about future system behaviour?
- How to use operational data, and what data to collect, when doing dynamic testing?
- How to deal with reproducibility, analysis, and diagnosis of failures when doing dynamic testing?

4.5 AI and Testing

Ever more systems will contain parts that use AI, which means that sooner or later the question will pop up on how to test such systems. Since AI works fundamentally different from traditional systems that contain explicitly coded algorithms, this may require new testing approaches and, probably even more, new bug analysis and diagnosis approaches. On the other hand, using AI in testing may provide new opportunities, e.g., for generating test cases or test analysis.

Both, testing systems with AI and using AI in testing are new and still in their infancy. Many things are unclear and a lot has to be investigated and learned. Many conferences and workshop in these areas appear, and papers are published, but because of the rapid developments in AI, many of these papers are volatile and have a short half-life. Since it is not to be expected that the AI movement will stop, corresponding testing questions will continue to pop up, and research in this area is needed.

4.6 Microservices

In Sect. 3.13, testing of a microservices architecture (MSA) was discussed, and it was argued that MSA has many of the testing challenges discussed in Chapter 3. So, MSA could be an interesting case study and playground for investigating the challenges and possible solutions. Moreover, MSA comes with a lot of structure, well-defined interfaces, and dynamic management of components (microservices), which makes it possible to also investigate some other aspects:

- it makes MSA suited for investigating methods for decomposition in building blocks, cf. Sect. 4.3;
- it facilitates applying formal modelling, formal verification techniques, and model learning (or test-based modelling, i.e., reverse engineering of a model from tests and observations) [77, 78].
- dynamic orchestration in MSA allows to create services whose purpose it is to test the ecosystem, thereby generalizing the passive monitors, i.e., testing as a microservice, which is part of the MSA.

4.7 More Research Topics

To conclude, a (non-exhaustive) number of additional topics and questions for possible applied research on testing for CPS are given. Some of these are related to the “Traditional Testing Challenges” of Sect. 2.2.

- Too often, testers first have to reverse engineer the specification or intentions of the developers before testing can really start: how to improve the specifications and requirements on which testing is based?
- How can coverage be expressed, how can it be determined, and what does it say about a test suite and about a SUT being tested?
- How can test-suite maintenance and maintenance of test automation be improved?
- Traditional testing is about comparing actual behaviour with expected behaviour. When going up in the test pyramid, and making abstractions, uncertainty increases: how to deal with uncertainty and nondeterminism in testing?
- Existing test suites can be huge, and are usually ever growing: how to select and prioritize from an existing test suite, when re-testing new versions or patches, and how to speed up this kind of testing?
- Test automation for higher levels of the test pyramid usually takes more effort and time than for the lower levels, because it is more domain specific and because there is an abstraction gap between the high-level concepts of the test cases and the low-level concepts of test execution: how to deal with bridging this gap and to facilitate high level test automation?
- How to determine when to stop testing, i.e., when is there enough confidence that the SUT is of good, or sufficient quality?
- Developing or selecting test cases can be based on usage behaviour of the system. This is called statistical model-based testing, or usage-profile (SUP) testing [79]. Such SUPs are obtained from usage logs of the system. It seems like an interesting approach to testing, but it is not that often used: how to improve SUP testing and remove its bottlenecks?
- Apart from testing, there are many other quality control and assurance techniques, see Sect. 2.2: how to combine these techniques in the most profitable way?
- BDD (behaviour-driven development) is currently quite a popular specification and testing approach in industry [80, 81]. It was developed in practice, without much input from research: how to improve BDD, give it a better underpinning, and connect research and practical developments?

- Testing has to be embedded in development processes: how to embed testing in current processes, such as agile, scrum, etc.?
- Quantum computing and quantum algorithms will pose new testing challenges, but, once available, it will most likely also offer new opportunities for improving and optimizing testing. At some time, testers will be confronted with the question to test quantum-based system. This might take a while, however, and it is less urgent than testing AI-based systems, but if quantum succeeds, one day the question will pop up: how to test quantum and how to use quantum in testing? Such questions are not specific for quantum computing, but apply equally to other emerging computing paradigms, e.g., neuromorphic computing.

5 Concluding Remarks

In this report, trends, challenges, and research directions for testing future generations of cyber-physical systems (CPS) were identified, with emphasis on testing the software part of CPS. Systematic testing is one of the most important ways of assessing and checking the quality of CPS software. Testing of CPS software is a challenging area for both (applied) researchers and test practitioners, and current and future developments will make it even more challenging.

The good news for testers is that there is a lot of interesting work to be done, and with the ever growing demand for better quality software in a context of continuously growing size, complexity, variability, speed of evolution, and connectivity of software, and with the emergence of new software paradigms and technologies, it is not likely that the importance of testing will diminish. The less good news is that it looks like software testing has difficulty in keeping pace with the speed, size, and complexity of software development. Ever more, testing turns out to be the bottleneck in releasing new features and products. This may hamper the fast and thorough testing of future generations of CPS, and thus slow down innovation. The software crisis is not over, yet [82].

More effective and efficient software testing is necessary, which requires new testing methodologies, automation, processes, and tools, for which, in turn, testing research and development are indispensable. Moreover, software testing is too often an afterthought: on product level when after designing and building the product without testing in mind, the product still has to be tested, and when new technologies are introduced in such a way that testing problems can be expected. Testers should be more involved in the forefront of development, assessing testability before a product is built, and giving back-pressure when a new technology or paradigm is being rolled out where testing problems are embedded. If high-quality products are desired, the motto should be: 'if you can't test it, don't build it'.

References

- [1] R.N. Charette. 'How Software Is Eating the Car'. In: *IEEE Spectrum* 58.6 (2021), pp. 22–29. URL: <https://spectrum.ieee.org/software-eating-car>.
- [2] S. Acur and T. Hendriks. *Vision and Outlook for Systems Architecting and Systems Engineering in the High Tech Equipment Industry*. TNO Public Report TNO 2024 R10542. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34642263/WXDoSj/TNO-2024-R10542.pdf>.
- [3] W. Oortwijn et al. *Vision and Outlook for System Evolution and Diversity*. TNO Public Report 2024 R10716. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34643767/xtSUpmKv/TNO-2024-R10716.pdf>.
- [4] R. Corvino et al. *The Future of Software Maintenance and Evolution*. TNO Public Report TNO 2026 R10278. The Netherlands: TNO, 2026. URL: <https://publications.tno.nl/publication/34645621/1AcLcouE/TNO-2026-R10278.pdf>.
- [5] J. Lukkien and J. Tretmans. *Vision and Outlook on Systems-of-Systems in the High-Tech Equipment Industry*. TNO Public Report 2025 R11685. The Netherlands: TNO, 2025. URL: <https://publications.tno.nl/publication/34645975/1bUdHobY/TNO-2025-R11685.pdf>.
- [6] L. Barbini, D. Bera and J. Marincic. *Vision and Outlook for Lifecycle Excellence in the High-Tech Equipment Industry*. TNO Public Report TNO 2025 R13007. The Netherlands: TNO, 2025. URL: <https://publications.tno.nl/publication/34645435/6r5BHvJF/TNO-2025-R13007.pdf>.
- [7] K.B. Akesson et al. *Software Evolution of Product Families in High-tech Equipment*. TNO Public Report TNO 2025 R10901. The Netherlands: TNO, 2025. URL: <https://publications.tno.nl/publication/34644776/fYZenVKu/TNO-2025-R10901.pdf>.
- [8] Capgemini-Sogeti. *World Quality Report – 17th Edition 2025-2026*. URL: <https://www.sogeti.nl/nieuws/quality-engineering/world-quality-report-2025-26>.
- [9] CPS Community Roadmap Report. Platforms4CPS Deliverable D2.2. 2018. DOI: <https://doi.org/10.5281/zenodo.10203896>.
- [10] KPMG UK Research and Benchmarking. *Software Testing Market and Insights Report*. KPMG Public, September 2024. KPMG LLP, 2024. URL: <https://assets.kpmg.com/content/dam/kpmgsites/uk/pdf/2024/08/software-testing-market-and-insights-report.pdf>.
- [11] E.A. Lee. 'Cyber Physical Systems: Design Challenges'. In: *11th IEEE Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing – ISORC*. 2008, pp. 363–369. DOI: <https://doi.org/10.1109/ISORC.2008.25>.
- [12] R. Alur. *Principles of Cyber-Physical Systems*. London, England: MIT Press, 2015.
- [13] E.A. Lee and S.A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. The MIT Press, 2016.
- [14] *Transactions on Cyber-Physical Systems*. New York, USA. URL: <https://dl.acm.org/journal/tcps>.
- [15] F.P. Brooks. *The Mythical Man-Month – Essays on Software Engineering*. Addison-Wesley, 1975.
- [16] P. Ammann and J. Offutt. *Introduction to Software Testing – Second Edition*. Cambridge University Press, 2017.

- [17] M.J. Harrold. 'Testing: A Roadmap'. In: *Procs. of the Conf. on The Future of Software Engineering*. ICSE'00. New York, USA: ACM, 2000, pp. 61–72. DOI: <https://doi.org/10.1145/336512.336532>.
- [18] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons Inc., 1979.
- [19] B. Beizer. *Software Testing Techniques*. Second Edition. Van Nostrand Reinhold, 1990.
- [20] M. Pezzè and M. Young. *Software Testing and Analysis – Processes, Principles, and Techniques*. John Wiley & Sons, Inc., 2008.
- [21] International Software Testing Qualifications Board. *Certified Tester Foundation Level Syllabus – V4.0.1*. Brussels, Belgium: ISTQB, 2024. URL: https://www.istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf.
- [22] International Organization for Standardization. *ISO/IEC 25010:2023. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model*. Geneva: ISO, 2023. URL: <https://www.iso.org/standard/78176.html>.
- [23] T. Klooster et al. *Software Security Testing Techniques and Tools*. TNO Public Memo TNO 2024 M10313. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34642661/70yCa9Lp/TNO-2024-M10313.pdf>.
- [24] E. de Gelder et al. *TNO StreetWise: Scenario-Based Safety Assessment of Automated Driving Systems*. TNO Public Report 2024 R10983. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34644772/XKcXFV7P/TNO-2024-R10983.pdf>.
- [25] J.H. Verriet. *Production Line Performance Optimisation*. TNO Public Report 2025 R11609. The Netherlands: TNO, 2025. URL: <https://publications.tno.nl/publication/34644850/zkz0CoiU/TNO-2025-R11609.pdf>.
- [26] J.J.M. Hooman and W. Tabingh-Suermondt. *Modeling and Analyzing Hardware-Software Interfaces with ComMA*. TNO Public Report 2024 R10720. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34642347/BuLgR4/TNO-2024-R10720.pdf>.
- [27] D. Graham et al. *Foundations of Software Testing: ISTQB Certification*. Thomson Learning EMEA, 2007.
- [28] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [29] E.W. Dijkstra. *Notes On Structured Programming – EWD249*. T.H. Report 70-WSK-03. Eindhoven, The Netherlands: Technische Hogeschool Eindhoven, 1969.
- [30] J. Tretmans. 'Model Based Testing with Labelled Transition Systems'. In: *Formal Methods and Testing*. Ed. by R.M. Hierons, J.P. Bowen and M. Harman. Vol. 4949. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 1–38. DOI: http://dx.doi.org/10.1007/978-3-540-78917-8_1.
- [31] J. Zander, I. Schieferdecker and P.J. Mosterman, eds. *Model-Based Testing for Embedded Systems*. CRC Press, 2012.
- [32] D. Bera. *Matala 2023 Technical Report*. TNO Public Report TNO 2023 R12766. The Netherlands: TNO, 2023. URL: <https://publications.tno.nl/publication/34642173/2tXRrg/TNO-2023-R12766.pdf>.
- [33] M.T.W. Schuts et al. 'Online Model-Based Testing by Reusing Multiple Design Models in an Industrial Setting'. In: *Journal of Object Technology* 24.2 (2025), pp. 1–14. DOI: <http://dx.doi.org/10.5381/jot.2025.24.2.a6>.
- [34] International Organization for Standardization. *ISO/IEC 9001:2015. Quality Management Systems – Requirements*. Geneva: ISO, 2015. URL: <https://www.iso.org/standard/62085.html>.

- [35] M.C. Paulk et al. 'Capability maturity model, version 1.1'. In: *IEEE Software* 10.4 (1993), pp. 18–27. DOI: <http://dx.doi.org/10.1109/52.219617>.
- [36] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, 1999.
- [37] M. Pol, R. Teunissen and E. van Veenendaal. *Software Testing, A Guide to the TMap Approach*. Addison Wesley, 2001.
- [38] T. Hendriks et al. 'A Virtual Test Platform for the Health Domain'. In: *Validation and Verification of Automated Systems: Results of the ENABLE-S3 Project*. Ed. by A. Leitner, D. Watzenig and J. Ibanez-Guzman. Springer Int. Pub., 2020. Chap. 21, pp. 297–320. DOI: https://doi.org/10.1007/978-3-030-14628-3_21.
- [39] W. Geurts, K. Wijbrans and J. Tretmans. 'Testing and Formal Methods — Bos Project Case Study'. In: *EuroSTAR'98: 6th European Int. Conference on Software Testing, Analysis & Review*. Munich, Germany: Aimware, Mervue, Galway, Ireland, 1998, pp. 215–229.
- [40] K. Pohl, G. Böckle and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [41] E.J. Weyuker. 'Testing Component-Based Software: A Cautionary Tale'. In: *IEEE Software* 15.5 (1998), pp. 54–59. DOI: <https://doi.org/10.1109/52.714817>.
- [42] J.L. Lyons. *Ariane 5: Flight 501 Failure*. Technical Report. The Inquiry Board, 1996. URL: <http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf>.
- [43] M.W. Maier. 'Architecting Principles for Systems-of-Systems'. In: *Systems Engineering* 1.4 (1998), pp. 267–284. DOI: [https://doi.org/10.1002/\(SICI\)1520-6858\(1998\)1:4<267::AID-SYS3>3.0.CO;2-D](https://doi.org/10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D).
- [44] P. van de Laar, J. Tretmans and M. Borth, eds. *Situation Awareness with Systems of Systems*. Springer, 2013. DOI: <http://dx.doi.org/10.1007/978-1-4614-6230-9>.
- [45] A. Zeller et al. *The Fuzzing Book*. Saarbrücken (D): CISPA Helmholtz Center for Information Security, 2019. DOI: <https://doi.org/10.60882/cispa.24614928>. URL: <https://www.fuzzingbook.org/>.
- [46] P. Godefroid. 'Fuzzing: Hack, Art, and Science'. In: *Communications of the ACM* 63.2 (2020), pp. 70–76. DOI: <https://doi.org/10.1145/3363824>.
- [47] C. Daniele, S.B. Andarzian and E. Poll. 'Fuzzers for Stateful Systems: Survey and Research Directions'. In: *ACM Comput. Surv.* 56.9 (2024). DOI: <https://doi.org/10.1145/3648468>.
- [48] W. Tabingh Suermondt et al. *Project Report: Sustainability Assessment of the Dutch High-Tech Industry 2023*. TNO Public Report 2024 R10421. The Netherlands: TNO, 2024. URL: <https://publications.tno.nl/publication/34642229/T5aXnZ/TNO-2024-R10421.pdf>.
- [49] P. Pelliccione et al. 'Beyond Connected Cars: A Systems of Systems Perspective'. In: *Science of Computer Programming* 191 (2020). DOI: <https://doi.org/10.1016/j.scico.2020.102414>.
- [50] F. Wotawa. 'On the Quest for Criticality Searching for Interactions that Matter When Hunting for Bugs'. In: *Go Where the Bugs Are: Essays Dedicated to Wolfgang Reif on the Occasion of His 65th Birthday*. Ed. by et al. Ernst G. Springer Nature Switzerland, 2025, pp. 356–373. DOI: https://doi.org/10.1007/978-3-031-92196-4_17.
- [51] P. Koopman and M. Wagner. 'Challenges in Autonomous Vehicle Testing and Validation'. In: *SAE Int. J. Trans. Safety* 4.1 (2016), pp. 15–24. DOI: <https://dx.doi.org/10.4271/2016-01-0128>.
- [52] J. Wang et al. 'Software Testing with Large Language Models: Survey, Landscape, and Vision'. In: *IEEE Trans. on Software Engineering* 50.4 (2024), pp. 911–936. DOI: <https://doi.org/10.1109/TSE.2024.3368208>.

- [53] M. Harman et al. 'Mutation-Guided LLM-based Test Generation at Meta'. In: *33rd ACM Int. Conf. on the Foundations of Software Engineering – FSE'25*. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 180–191. DOI: <https://doi.org/10.1145/3696630.3728544>.
- [54] P. Jamshidi et al. 'Microservices: The Journey So Far and Challenges Ahead'. In: *IEEE Software* 35.3 (2018), pp. 24–35. DOI: <https://doi.org/10.1109/MS.2018.2141039>.
- [55] N. Dragoni et al. 'Microservices: Yesterday, Today, and Tomorrow'. In: *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Springer Int. Publishing, 2017, pp. 195–216. URL: https://doi.org/10.1007/978-3-319-67425-4_12.
- [56] J. Fritzsche et al. 'Adopting Microservices and DevOps in the Cyber-Physical Systems Domain: A Rapid Review and Case Study'. In: *Software: Practice and Experience* 53.3 (2023), pp. 790–810. DOI: <https://doi.org/10.1002/spe.3169>.
- [57] B. Li et al. 'Enjoy your Observability: An Industrial Survey of Microservice Tracing and Analysis'. In: *Empirical Software Engineering* 27.1 (2022). DOI: <https://doi.org/10.1007/s10664-021-10063-9>.
- [58] A. Benveniste et al. 'Contracts for System Design'. In: *Foundations and Trends in Electronic Design Automation* 12.2–3 (2018), pp. 124–400. DOI: <https://dx.doi.org/10.1561/10000000053>.
- [59] D. Bera, K. Traganos and M. Vicari. *Matala 2025: Model-Based Testing of Systems in Context*. TNO Internal Report TNO 2026-18003. The Netherlands: TNO, 2025.
- [60] M. Bruntink and A. van Deursen. 'An Empirical Study into Class Testability'. In: *Journal of Systems and Software* 79.9 (2006), pp. 1219–1232. URL: <http://dx.doi.org/10.1016/j.jss.2006.02.036>.
- [61] M. Harman et al. 'Testability Transformation – Program Transformation to Improve Testability'. In: *Formal Methods and Testing*. Ed. by R.M. Hierons, J.P. Bowen and M. Harman. Vol. 4949. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 320–344. URL: http://dx.doi.org/10.1007/978-3-540-78917-8_11.
- [62] E.A. Lee. 'What Is Real Time Computing? A Personal View'. In: *IEEE Design & Test* 35.2 (2018), pp. 64–72. DOI: <https://doi.org/10.1109/MDAT.2017.2766560>.
- [63] T. Hendriks et al. *MBSE in the High-Tech Equipment Industry – MBSE-Study phase 2 of TNO-ESI and Partners: Key MBSE Topics and Approaches*. TNO Public Report 2025 R11769. The Netherlands: TNO, 2025. URL: <https://publications.tno.nl/publication/34644955/9a01yNYM/TNO-2025-R11769.pdf>.
- [64] SysML – Systems Modeling Language. Object Management Group. URL: <https://sysml.org/>.
- [65] T. Hulshof. 'Using SysML for Model-Based Testing'. MA thesis. Nijmegen, The Netherlands: Institute for Computing and Information Sciences, Radboud University, 2022. URL: <https://www.cs.ru.nl/masters-theses>.
- [66] D. van der Wal. 'What is ... the Point? – Single-Input-Change Testing a EULYNX Controller'. PhD thesis. Enschede, The Netherlands: University of Twente, 2025. DOI: <https://doi.org/10.3990/1.9789036565042>.
- [67] *Eclipse CommaSuite*. Eclipse Foundation. URL: <https://eclipse.dev/comma>.
- [68] J. Tretmans and P. van de Laar. 'Model-Based Testing with TorXakis – The Mysteries of Dropbox Revisited'. In: *Central Eur. Conf. on Information and Intelligent Systems – CECIIS*. Ed. by V. Strahonja, D. Hertweck and V. Kirinić. Faculty of Organization and Informatics, University of Zagreb, Varaždin, Croatia, 2019, pp. 247–258. URL: <http://archive.ceciis.foi.hr/app/public/conferences/2019/Proceedings/QSS/QSS3.pdf>.

- [69] T.C. Ruys and M. van der Bijl. ‘Effective Model-Based Testing’. In: *Principles of Verification: Cycling the Probabilistic Landscape : Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Ed. by et al. Jansen N. Springer Nature Switzerland, 2025, pp. 234–251. DOI: https://doi.org/10.1007/978-3-031-75778-5_11.
- [70] S. Mohacsi and M. Felderer. ‘AI-Based Enhancement of Test Models in an Industrial Model-Based Testing Tool’. In: *IEEE Int. Conf. on Software Analysis, Evolution and Reengineering – SANER 2021*. 2021, pp. 636–638. DOI: <https://dx.doi.org/10.1109/SANER50967.2021.00080>.
- [71] R. Mondal et al. ‘EYWA: Automating Model Based Testing using LLMs’. In: *23rd USENIX Symp. on Networked Systems Design and Implementation – NSDI 26*. USENIX Association, 2026, pp. 777–800. URL: <https://www.usenix.org/conference/nsdi26/presentation/mondal>.
- [72] S. Apel et al. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer Int. Publishing, 2013. DOI: <https://doi.org/10.1007/978-3-642-37521-7>.
- [73] T. Berger et al. ‘What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines’. In: *19th Int. Conf. on Software Product Lines – SPLC’15*. New York, USA: ACM, 2015, pp. 16–25. DOI: <https://doi.org/10.1145/2791060.2791108>.
- [74] *TiCToC – Testing in Times of Continuous Change*. NWO-TTW OTP project. URL: <https://tictoc.cs.ru.nl>.
- [75] G. van Cuyck, L. van Arragon and J. Tretmans. ‘Compositionality in Model-Based Testing’. In: *IFIP-ICTSS – 35th Int. Conf. on Testing Software and Systems*. Ed. by S. Bonfanti, A. Gargantini and P. Salvaneschi. Vol. 14131. Lecture Notes in Computer Science. Springer Nature Switzerland AG, 2023, pp. 202–218. DOI: https://doi.org/10.1007/978-3-031-43240-8_13.
- [76] G. van Cuyck, L. van Arragon and J. Tretmans. ‘Testing Compositionality’. In: *Science of Computer Programming* 255 (2027), p. 103537. DOI: <https://doi.org/10.1016/j.scico.2026.103537>.
- [77] Vaandrager, F. ‘Model Learning’. In: *Communications of the ACM* 60.2 (2017), pp. 86–95. DOI: <https://doi.org/10.1145/2967606>.
- [78] Hendriks, D. ‘Model Inference and Comparison for Software Evolution in Large Component-Based Systems’. PhD thesis. Nijmegen, The Netherlands: Radboud University, 2024. DOI: <https://doi.org/10.54195/9789493296640>.
- [79] J.H. Poore et al. ‘Automated Statistical Testing for Embedded Systems’. In: *Model-Based Testing for Embedded Systems*. Ed. by J. Zander, I. Schieferdecker and P.J. Mosterman. CRC Press, 2012, pp. 111–146.
- [80] G. Nagy and S. Rose. *The BDD Books – Discovery: Explore Behaviour Using Examples*. Leanpub Publishing, 2018. URL: <http://leanpub.com/bddbooks-discovery>.
- [81] T. Zameni et al. ‘From BDD Scenarios to Test Case Generation’. In: *19th Int. Workshop on Advances in Model-Based Testing – A-MOST 2023*. Ed. by F. Lorber, C. Secleanu and U.C. Türker. IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW). IEEE CS, 2023, pp. 36–44. DOI: <https://doi.org/10.1109/ICSTW58534.2023.00019>.
- [82] W.W. Gibbs. ‘Software’s Chronic Crisis’. In: *Scientific American* 271.3 (1994), pp. 72–81.

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl