The growing role of software in medical processes emphasizes its critical function in ensuring safety during diagnosis, decision-making, and device operation. Recent efforts focus on establishing robust software life cycles and adhering to certification standards through thorough verification and validation.
This work explores methodologies for developing medical devices, highlighting the efficacy of model-based system engineering (MBSE) for compliance. Furthermore, Combinatorial Interaction Testing proves beneficial for complex systems, as demonstrated in case studies like the MVM ventilator, Pill-Box, and PHD protocol. Finally, for medical software integrating artificial intelligence (AI), alternative methods based on neural network robustness analysis are proposed for validation.

Andrea Bombarda obtained his PhD in Engineering and Applied Sciences (35th Cycle) at the University of Bergamo. During his university studies in Computer Engineering, he carried out research experiences in research institutions such as the University of Texas at Arlington. After earning his PhD, he currently works as a Research Associate at the University of Bergamo in the context of the ANTHEM project.

ISBN: 978-88-97413-86-8
DOI: 10.13122/978-88-97413-86-8

Andrea Bombarda S.Q.A. FOR MEDICAL SYSTEMS 60

Collana della Scuola di Alta Formazione Dottorale
- 60 -

Andrea Bombarda

A COMPREHENSIVE APPROACH for Software Quality Assurance for Medical Systems

UNIVERSITÀ DEGLI STUDI DI BERGAMO

2024

Collana della Scuola di Alta Formazione Dottorale

- 60 -

Collana della Scuola di Alta Formazione Dottorale

Diretta da Paolo Cesaretti

Ogni volume è sottoposto a *blind peer review*.

ISSN: 2611-9927

Sito web: https://aisberg.unibg.it/handle/10446/130100

Andrea Bombarda

A COMPREHENSIVE APPROACH for Software Quality Assurance for Medical Systems

Università degli Studi di Bergamo

2024

A Comprehensive Approach for Software Quality Assurance
for Medical Systems / Andrea Bombarda. – Bergamo :
Università degli Studi di Bergamo, 2024.
(Collana della Scuola di Alta Formazione Dottorale; 60)

ISBN: 978-88-97413-86-8
DOI: 10.13122/978-88-97413-86-8

Progetto grafico: Servizi Editoriali – Università degli Studi di Bergamo
© 2018 Università degli Studi di Bergamo
via Salvecchio, 19
24129 Bergamo
Cod. Fiscale 80004350163
P. IVA 01612800167

https://aisberg.unibg.it/handle/10446/280211

Acknowledgments

I would like to thank my supervisor Prof. Angelo Gargantini, for his guidance, support and help during my Ph.D. Moreover, I would like to thank Prof.ssa Elvinia Riccobene, Dr. Paolo Arcaini, Ing. Silvia Bonfanti, and Prof.ssa Patrizia Scandurra for their continuing and patient support. A lot of the work presented in this book would not have been possible without their collaboration.
A special thanks to my girlfriend Alessandra, my parents, my brother and the whole "Bombotti" family for their support and encouragement.
I would like to extend my thanks to Prof. Jeff Lei, from UTA (University of Texas at Arlington), and all his Ph.D. students for having welcomed me in their university during many visiting periods.
Finally, I wish to thank the many kind people (researchers and not) with whom I had the pleasure to interact during the journey to produce this book. I would like to thank the University of Bergamo for making it possible.

I would like to dedicate this book to all those who have believed in me

Table of Contents

Introduction

Many people interact with medical devices every day, from the simplest ones, such as pillboxes that remind the patients to take their pills on the correct day, to the most complex ones such as diagnostic X-ray equipments or mechanical ventilators in intensive care units (ICUs). Some of these devices were traditionally based on mechanical and electrical components, and all hardware parts were tested to guarantee their correct functioning. However, with the development of the technologies, the software has increasingly become a prominent and critical part of these devices and this has led to the definition of new kinds of medical devices: Programmable Electronic Medical Systems (PEMS). These safety-critical systems combine hardware and software to implement their functionalities, which affect people's health, and, in case of malfunctions, they can seriously compromise human safety leading to injuries or even death. For this reason, software that controls or interacts with these devices must be developed through rigorous processes that aim to ensure its safety, reliability, and quality, and certified.

To certify the quality of medical systems, several standards have been proposed (see Sect. 1.2), but the majority of them do not deal with the software part of PEMS and limit only to the hardware part. The only two standards or guidelines that include references to software are IEC 62304 [64] and the FDA Guidelines [83]. However, in both documents, only general concepts about the activities that must be performed for each medical software are given, and no suggestions on the software life cycle to be used or methods for software validation and verification are presented.

For this reason, this book presents and analyzes several approaches compliant with the regulations and standards for PEMS certification, aiming at improving or assuring the quality of medical software and systems. The majority of the experiences and experiments contained in this book are based on the Mechanical Ventilator Milano (MVM) case study (see Chapter 2), which is a mechanical ventilator for intensive care units I contributed to develop, test, and certify during the first wave of the COVID-19 pandemic in Italy. From this experience, several lessons learned and general guidelines for developing medical systems under emergency, while still maintaining compliance with the certification standards, have been derived. Further details on these guidelines are reported in Chapter 3.

Model-based testing and model-driven development have proven to be suitable for complying with the certification standards, since they allow producing documents in a formal manner, which is easily verifiable, and guarantee the required traceability. In this book, I present the application of formal methods, and in particular of the ASM formalism using the ASMETA framework [7], to the MVM

case study. It has proved to be suitable for system modeling, validation, verification and automatic code generation. Moreover, I present how using ASMETA can aid in automatically deriving test cases to be used on the actual implementation of the ventilator. In order to assure the generalizability of the results, this book presents the application of model-based techniques to other systems, such as a Pill-Box and the PHD protocol, used for allowing the communication between medical devices.
A limitation discovered during the application of formal methods, such as ASMETA, to the analyzed case studies for generating test cases, is the significant amount of tests and time required for their generation or execution. For this reason, this book presents the concept of Combinatorial Interaction Testing applied to medical software and systems and a way to compare combinatorial test generators. This aims at choosing the best-fit generator which produces fewer test cases or limits the time required for the test generation process, depending on the complexity of the system.
Although certification standards describe which activities must be performed for each medical software, no complete indication is yet available on how to certify medical systems containing AI-based components. The only request made by the certifying authorities is that AI algorithms must show to be at least as safe and effective as another similar legally marketed, which is not based on AI. For this reason, in this book, I propose the concept of robustness for neural networks (used both as classifiers or estimators) when they are used in the medical domain. Using this measure, a medical system can be certified by showing that on the extensive data set used during testing activities, it has proved to be at least as reliable as other devices already on the market.

Research questions, objectives, and book structure

The goal of this book is to apply state-of-the-art methods, or devise and design new ones, for increasing or assuring the quality of medical software and systems. In the following, I report the research questions (RQs) I answer through the chapters of this book.

RQ1: What is the state-of-the-art in the definition of quality for medical software? This is shown in Chapter 1 in which the concept of quality for software, the definition of medical device, some legal aspects about the liability for non-quality medical systems, and standards and regulations for the certification of medical software are analyzed. In particular, two documents are presented in detail: the standard IEC 62304 [64] and the FDA general principles [83]. The former is an international standard, which describes the life cycle activities of the software development without giving any indication on the method to be applied. The latter provides the guidelines for the validation and verification of the medical software.

RQ2: Are there any empirical guidelines for developing medical software? During the first wave of the COVID-19 pandemic, I contributed together with my research group and other researchers and practitioners in the development, testing, and certification of a real medical system, the Mechanical Ventilator Milano (see Chapter 2 for more details about the device). Chapter 3 presents lessons learned and guidelines derived from this effort, especially in the context of an emergency.

RQ3: Are ASMs applicable to support the development of software for medical devices? Chapter 4 presents the ASMETA framework and the set of tools included in the framework to support the user during the development and the analysis of formal specifications using ASMs. Furthermore, in order to practically evaluate the applicability of the ASMETA framework to medical software, in Chapter 5 the application of the ASMETA framework to real case studies, such as the MVM, a Pill-Box, and the PHD communication protocol, is reported.

RQ4: Considering that combinatorial testing is often used for reducing the complexity of testing medical software, is there a way to make the test generation process faster? In Chapter 6, I present the concept of Combinatorial Interaction Testing (CIT) applied to safety-critical systems, such as medical software. In particular, the difference between several test generators, in terms of test suite size and generation time is analyzed, and two methodologies for reducing the generation time and, in general, the cost of testing are presented and used for generating test suites for the MVM case study.

RQ5: Are there methods to validate NN-based medical software? Chapter 7 presents the novel concept of neural network robustness against input alterations, both for classifiers and estimators. This measure can be used for increasing the confidence in neural network models used in safety-critical domains. Furthermore, I present ROBY, a tool for automatic robustness computation for neural network classifiers, and ASAP, a method for reducing the time required for this evaluation. Moreover, in Chapter 8 the robustness computation is performed on two different case studies in the medical domain, namely the breast cancer classification and the blood PO_2 estimation.

This book reports the results of an ongoing research activity carried out by scientific groups I was part of during my Ph.D. Some parts of the presented work have been developed by co-authors of the papers published, but they are reported in this book for completeness and to ensure that the reader is able to fully understand the content.

Chapter 1. State of the art

In this chapter, I present the main reasons behind the research activities reported in this book. In particular, the concept of quality for software is presented and discussed in Sect. 1.1, independently from its application in the medical software field. Sect. 1.2 presents the current state of the art in terms of standards and regulations to be followed for certifying medical software, while Sect. 1.3 defines what can be considered a medical device. Then, the responsibilities in case of failure related to the non-quality of a medical system are discussed in Sect. 1.4, based on the current state-of-the-art in the Italian, European and International regulations. Finally, Sect. 1.5 concludes the chapter.

1.1 Quality of software

ISO defines software quality as the "capability of a software product to conform to its requirements". To be more specific, in the context of software engineering, the concept of *software quality* refers to two related but different aspects:

- *Functional quality*, or *external quality*, which reflects how well the software complies with a given design, usually based on its requirements or specifications. It is generally decomposed into correctness, reliability, integrity, and usability.
- *Structural quality*, or *internal quality*, which depends on how the software meets non-functional requirements that support the delivery of the functional requirements, such as re-usability, portability, efficiency, or maintainability.

As reported in the list above, several factors influence the quality of the software as described, for example, in the McCall's quality model [130], the following properties contribute in defining software quality:

- *Correctness*, i.e., the extent to which a program complies with its specifications;
- *Reliability*, i.e., the system's ability not to fail;
- *Integrity*, i.e., the protection of the program from unauthorized access;
- *Usability*, i.e., the ease of the use of the software;
- *Re-usability*, i.e., the ease of reusing software in a different context;
- *Portability*, i.e., the effort required to transfer a program from one environment to another;

- *Efficiency*, i.e., the way in which the software uses the resources, e.g., processor time or storage;
- *Maintainability*, i.e., the measure of the effort required to locate and fix a fault in the program within its operating environment.

Software quality is motivated by at least two main perspectives: *risk management* and *cost management*. The former, especially in safety-critical contexts, considers the fact that software errors can cause human fatalities (see Sect. 1.1.1 for some examples in the context of medical systems) which have been historically caused by poorly designed user interfaces or direct programming errors. The latter considers that a software product or service governed by good software quality costs less to maintain, is easier to understand, and is more cost-effective when it has to be changed in response to pressing business needs.

1.1.1 Examples of non-quality accidents in medical systems

The literature related to the development of medical software and systems reports several examples of non-quality accidents [119]. Probably, the best-known example is the one of Therac-25 [120] between 1985 and 1987, a medical electron accelerator that was involved in six massive radiation overdoses, hundreds of times greater than normal. As a result, several people died and others were seriously injured. Investigations on device software have identified several causes for this malfunction, among which there are the following: (I) the source code was never independently reviewed, (II) the hardware has never been tested with the final software until the device was assembled at the hospital, (III) several error messages were not explained in the user manual and no indication about these errors and possible threats to patient safety was mentioned, (IV) the software set a flag variable by incrementing it, rather than by setting it to a fixed non-zero value, leading to occasional arithmetic overflows that cause the software to bypass safety checks, (V) software engineers had reused software from the Therac-6 and Therac-20, which used hardware interlocks that masked their software defects and were not present in Therac-25, (VI) no verification on the correct behavior of the sensors was performed by the software before operating.

Similar is the case of the therapy planning software created by Multidata Systems International [45], which miscalculated the appropriate dose of radiation for patients undergoing Cobalt-60 radiation therapy in Panama. Multidata's software allowed a radiation therapist to draw on a computer screen the placement of metal shields called “blocks” designed to protect the healthy tissue of patients from radiations. However, the software only allowed technicians to use four shielding blocks, and the Panamanian doctors wished to use five. The doctors discovered that they could trick the software by

drawing all five blocks as a single large block with a hole in the middle. Nevertheless, what the doctors did not realize is that the Multidata software gave different answers in this configuration depending on how the hole was drawn: if the hole was drawn in one direction, then the correct dose of radiation was calculated, but if it was drawn in another directions the software recommended twice the necessary exposure.
However, not all problems are related only to the software embedded in medical devices. For example, in [19], the authors report the results of analyses conducted over more than 30 thousand patients from the US, where 1 out of 5 of them reported finding a mistake in their Electronic Health Records (EHR), sometimes caused by bugs in the IT system, and 40% perceived the mistake as serious. Older and sicker patients were twice as likely to report a serious error compared with younger and healthier patients, indicating important safety and quality implications since not being able to identify in time the correct outcome of a medical exam may lead to serious consequences, which could be easily avoided if the EHR were correct.

1.2 Standards and regulations for the certification of medical software

Due to the safety-critical nature of medical devices, many institutions have provided standards or guidelines for their certification. Examples of these organizations are the IEC (International Electrotechnical Commission), ISO (International Organization for Standardization), European Union, FDA (Food and Drug Administration), ANSI (American National Standards Institute), UNI (Italian Organization for Standardization), DIN (German Institute for Standardization), and CEI (Italian Electrotechnical Commission).
In the following, the main standards, norms, and regulations produced by the organizations listed above are reported:

- The ISO 13485 [105], approved also by CEI and UNI, describes the requirements on how to apply the ISO 9001 [101] for quality system management, for design and development, installation, and maintenance of medical devices. It has been recently adopted also by FDA, which has harmonized its requirements to those of the ISO 13485 standard.

- The ISO 14971 [104] specifies terminology, principles, and a process for risk management of medical devices, including *software as a medical device* and in vitro diagnostic medical devices. It is recognized both by FDA (for the US region) and by European certification authorities.

- The European regulation UE 2017/745 [1] defines a new device classification and device scope, stricter oversight of manufacturers by Notified Bodies, the introduction of the "Person Responsible for Regulatory Compliance" (PRRC) and the economic operator concept, the requirement of UDI marking for devices, Eudamed[1] registration, and increased post-market surveillance activities. Being a European regulation, it is not requested by FDA for the certification in the US.

- The IEC 62304 [64] defines the phases to be followed during the life cycle of medical device software, depending on the risk classification of the device (see Sect. 1.2.1 for a more detailed description). In general, this is the most used standard, and it is recognized by almost all medical software certification authorities in the world.
- The IEC 61508-3:2010 [65] defines software requirements for functional safety of electrical, electronic, or programmable electronic safety-related systems. No formal requirement to use this standard is given by none of the certification authorities, but its adoption is strongly advised.
- The IEC 60601-1:2005 [63] contains requirements concerning basic safety and essential performance that are generally applicable to medical electrical equipment. It is recognized both by FDA (for the US region) and by European certification authorities.
- The "General Principles of Software Validation" produced by FDA [83] defines several concepts aiming at guiding the software validation and verification activities, which are required to be performed throughout the software development life cycle (see Sect. 1.2.2 for a more detailed description). These guidelines have been proposed by FDA and, thus, are required only in the United States. However, the principles proposed fit with those of the other main certification standards (e.g., IEC 62304) which are, instead, required in Europe.

Note that none of these standards or regulations prescribes specific methodologies in which all required activities must be performed.

1.2.1 IEC 62304: Medical Software Development Life Cycle

The IEC 62304 [64], named "Medical Device Software – Software Life Cycle Process", has emerged as the main standard for the management of the software development life cycle for medical devices, it is adopted by the European Union and the United States, and approved as an international standard both by IEC and ISO. Moreover, it is adopted by several committees and institutions, such as the CENELEC (European Committee for Electrotechnical Standardization), ANSI (American National

[1]Eudamed is an online database, created by the European Commission, providing a living picture of the lifecycle of medical devices that are made available in the European Union.

Standard Institute), FDA (Food and Drug Administration) of the United States for use in premarket submissions, SFDA (State Food and Drug Administration) of China and Japan Industry.
The standard defines common guidelines for medical device software life cycle processes, but does not impose a specific life cycle model or give guidelines on how the software should be verified and validated.
The major contribution of the IEC 62304 standard is the definition of three different *safety classes*, based on the potential to create an injury to patients by the device:

- *Class A*: medical software, or software components, that cannot cause injury or damage to health;
- *Class B*: medical software, or software components, that can cause non-serious injury;
- *Class C*: medical software, or software components, that can cause death or serious injury.

In particular, the standard classifies as *serious injury* an injury that "directly or indirectly is life-threatening, can result in permanent impairment of a body function or permanent damage to a body structure, or necessitates medical or surgical intervention to prevent permanent impairment of a body function or permanent damage to a body structure".

Processes and tasks per safety class

The IEC 62304 standard defines the processes and tasks (among those reported in Fig. 1) that are mandatory depending on the class in which the software under development is classified. In general, the higher the safety class (B and C), the higher the number of additional tasks and the more detailed needs to be the documentation. Tab. 1 reports the activities that are required by each safety class and their mapping on the activities as they are numbered in the standard.

Software development planning (5.1) The first step of the software development life cycle consists in defining the life cycle model to be used. In this plan, one should address: 1. the process to be used during the development of the software; 2. the deliverables of each activity and the tasks to be performed; 3. the traceability system among system requirements, software requirements, test and risk control activities; 4. the software configuration and software problem resolution strategy.

Software requirements analysis (5.2) During the software requirements analysis, functional and non-functional requirements are defined. Moreover, it is required to measure the risk of potential

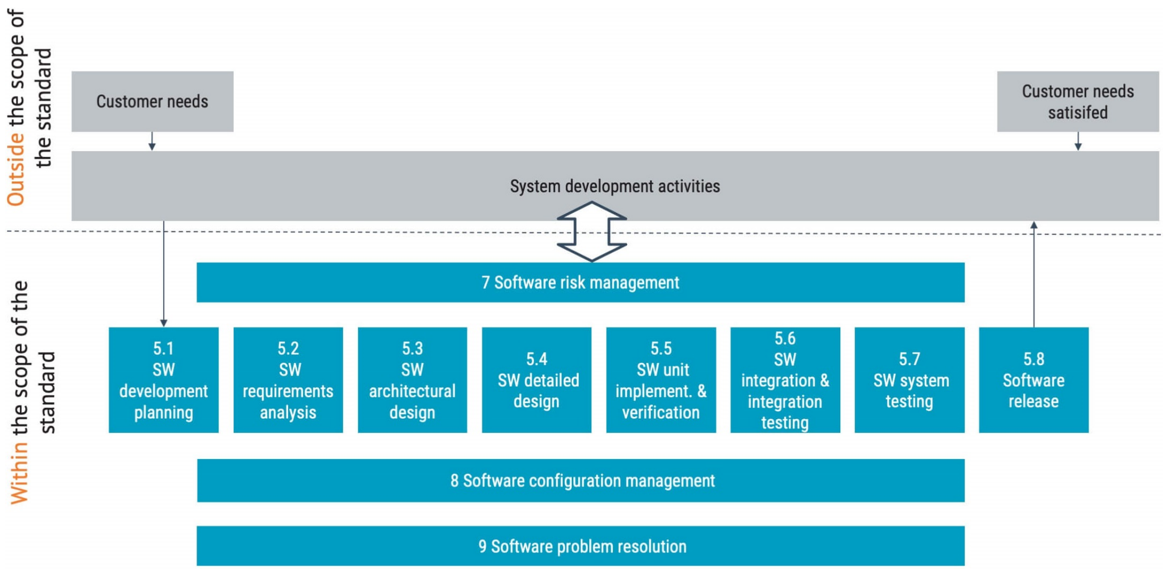

Outside the scope of the standard
Customer needs
Customer needs satisifed
System development activities
Within the scope of the standard
7 Software risk management
5.1 SW development planning
5.2 SW requirements analysis
5.3 SW architectural design
5.4 SW detailed design
5.5 SW unit implement. & verification
5.6 SW integration & integration testing
5.7 SW system testing
5.8 Software release
8 Software configuration management
9 Software problem resolution

Figure 1: Activities within and outside the scope of the IEC 62304 standard.

Activity	Class A	Class B	Class C
Software development planning (5.1)	X	X	X
Software requirements analysis (5.2)	X	X	X
Software architecture (5.3)		X	X
Software detailed design (5.4)			X
Software unit implementation (5.5)	X	X	X
Software unit verification (5.5)		X	X
Software integration (5.6)		X	X
Software integration testing (5.6)		X	X
Software system testing (5.7)	X	X	X
Software release (5.8)	X	X	X
Software maintenance process (6)	X	X	X
Software risk management process (7)		X	X

Table 1: IEC 62304 software development process – Activities required by each safety class.

defects in every requirement. At every update, the manufacturer should verify software requirements to avoid contradiction and ambiguity, and re-evaluate the risks previously identified.

Software architecture (5.3) During the definition of the software architecture, it is required to describe the software structure, identify the software elements, specify functional and performance requirements for the software elements, identify software elements related to risk control, and verify the software architecture with respect to the previously defined software requirements.

Software detailed design (5.4) Once the software architecture has been identified, it should be detailed into software units and, for each unit, a detailed design should be provided.

Software unit implementation and unit verification (5.5) Each software unit should be implemented and tested. During the software unit verification, the manufacturer shall verify the source code w.r.t. the previously produced documents, and the results of the verification activities shall be systematically documented.

Software integration and integration testing (5.6) During this phase, each software unit is integrated with the others on the basis of the integration plan and the final system is verified. As for the unit testing activities, the manufacturer shall systematically document the results of the verification activities.

Software system testing (5.7) Once the system has been fully integrated, the tests are performed on the system as a whole, w.r.t. the system requirements. This activity has to be repeated after each change in the software.

Software release (5.8) When releasing the software, the device manufacturer shall demonstrate that the software has been correctly and extensively validated and verified. In case of anomalies, the manufacturer has to document and establish a process to resolve them.

Software maintenance process (6) During this activity, the manufacturer shall describe the procedures used for the maintenance process. These procedures have to be applied when intervening on the product that is in use.

Software risk management process (7) The last step that the manufacturer must perform is the analysis of risk. This analysis aims to identify causes that could contribute to an unsafe situation and should be performed during the whole software life cycle.

1.2.2 FDA Guidelines: General Principles of Software Validation

The Food and Drug Administration is a federal agency of the United States and is responsible for protecting and promoting public health through the control and supervision of food safety, tobacco products, dietary supplements, medications, vaccines, biopharmaceuticals, blood transfusions, medical devices, electromagnetic radiation emitting devices, cosmetics, animal foods, and veterinary products. In terms of medical software, the *General Principles of Software Validation* [83] defines several concepts that can be used as guidance for software validation and verification activities, which are required to be performed throughout the software development life cycle. Moreover, these concepts are not limited to software considered as a medical device but also to the software used to produce medical devices and software used in the quality management processes of the manufacturer. In the following, the main principles reported in the document are listed:

- Validation and verification activities should use as a baseline the *software requirements specification* document;
- Developers should apply a set of methods and techniques for preventing the introduction of defects during the software development process, and not only try testing the software after it is completely written;
- Software validation activities should be planned early in the development process and performed during all the software life cycle;
- Software life cycle should contain documents necessary for supporting the software validation and software engineering tasks;
- Software life cycle should contain verification and validation tasks aimed at guaranteeing the correct functioning of the software during its intended use;
- Software validation and verification processes should be defined and controlled through the use of a *plan*, and executed using a set of defined *procedures*;
- When the software is changed, validation analysis should be conducted not just for the individual change, but also to determine the impact on the entire software system, to demonstrate that the change has not altered the system behavior;
- Validation effort and coverage should be based on software complexity and safety risk. In this way, if the risk increases additional validation activities should be added to guarantee the coverage of all risks;
- Validation and verification activities should be conducted using the quality assurance precept of the independence of review to guarantee the quality of software. These activities should be assigned to staff members that are not involved in the design or implementation of software;
- No validation principle is mandatory, but the device manufacturer retains ultimate responsibility for demonstrating that the software has been correctly validated.

Note that the FDA guidelines do not classify medical devices in safety classes and, therefore, each device manufacturer should perform a combined evaluation using both these guidelines and the IEC 62304 standard to identify the activities to be performed.

1.3 The definition of "medical device"

All the standards mentioned in the previous section are based on the same definition of "medical device" and, consequently, of "medical software". In particular, the following are considered medical devices:

- any instrument, appliance, software, or substance used alone or in combination with other devices during one of the activities related to the medical field;
- any software intended by the manufacturer for use specifically for diagnostic or therapeutic purposes, used for assuring the correct functioning of medical instruments;
- any software intended by the manufacturer to be used on human beings for diagnosis, prevention, control, therapy, or alleviation of a disease;
- any software for diagnosis, control, therapy, mitigation, or compensation for an injury or handicap;
- any appliance intended to be used as a replacement or modification of the anatomy or a physiological process.

Given these definitions, it is clear that the software of a medical device can also be considered a medical device itself. In the literature, this aspect is generally defined *software as medical device* (SaMD). Moreover, electronic devices that integrate software used in medical processes are normally defined *programmable electrical medical systems* (PEMS).

1.4 Legal aspects of the liability for non-quality medical systems

Complying with standards and regulations not only aims to obtain a medical device with higher quality, but also to avoid legal consequences deriving from a possible malfunction of the device for the manufacturer. In fact, current guidelines and regulations (see Sect. 1.2 for further details) consider the device manufacturer the one who assumes all the possible responsibilities and bears any administrative sanctions deriving from any malfunctioning of the produced medical device.

All the standards require producing a consistent number of documents, and it is justified by the fact that all these documents can be used at a later time for a re-verification in case the marketed medical device causes any damage to a human being. Indeed, the legal consequences deriving from a possible malfunctioning of a medical device for its manufacturer can be limited or avoided if in the documentation it is clear that:

- The device is not marketed yet;
- The defect causing the damage did not exist when the manufacturer started marketing its product (so, it is of paramount importance to maintain in the documents all the references of all the validation tests performed);
- The defect causing the damage cannot be avoided if the manufacturer must comply with some regulations concerning the application field;
- The defect causing the damage could not have been discovered when the manufacturer started marketing its product due to the limited scientific knowledge in the state-of-the-art at that time;
- The defect causing the damage is not due to the product itself, but entirely to the product that embedded the produced device.

1.5 Conclusion

In this chapter, the concept of quality for software in a broader way, and in specific for medical devices, has been presented, and standards and guidelines adopted for the certification of medical device software have been identified. The standard IEC 62304 identifies the step of the software development life cycle, without reference to a specific life cycle. Companies that want to certify their software have to map the applied process for software development with the totality of the clauses of the IEC 62304 standard. Once the documents are available, they are submitted to the organizations responsible for certifying the software that evaluate the documentation and provide the certification or request for changes to fit the presented process with the standards.
This certification process has to be followed by producers not only for getting the certification and the authorization for marketing the products, but also for assuring the quality of the produced medical device and for protecting themselves w.r.t. possible malfunctioning consequences.
The effort of adhering to IEC 62304 and certifying a medical device, together with the lessons learned from this process are presented, for the MVM case study (see Chapter 2), in this book in Chapter 3.

Part I

MVM: A mechanical ventilator for ICUs

Chapter 2. The MVM case study

In this chapter, one of the main case studies for the research activities discussed in this book is presented. Here, I introduce the Mechanical Ventilator Milano (MVM), which is a project to which I have contributed during the first phases of the emergency related to the COVID-19 pandemic in Italy, especially for the process of software testing and its certification as a medical device. The chapter is based on [2] and is structured as follows. Sect. 2.1 introduces the history of the MVM device, together with the reasons that motivated the work on it. Sect. 2.2 reports some of the medical considerations about the conditions and treatments that are generally performed in patients with COVID-19, while Sect. 2.3 describes the design of the device. In Sect. 2.4 the functioning of the MVM and its operating modes are presented, while the hardware and software configuration of the ventilator is introduced in Sect. 2.5. Finally, Sect. 2.6 and Sect. 2.7, respectively, present the testing procedures (not limited only to the software components) executed on the final device and conclude the chapter.

2.1 Introduction

Mechanical ventilators are necessary tools in every modern intensive care unit (ICU). The type and intensity of ventilation support required by a patient vary over the course of treatment. Therefore, modern mechanical ventilators are versatile and adapt to patient needs. Commercially available devices control volume, pressure, or gas flow, and the breathing cycle timing. They support both patients who cannot breathe, and who can still trigger a mechanical cycle by a spontaneous inspiratory effort [174]. Today's mechanical ventilators are complex machines, composed of many specialized components and implementing several ventilation modes [109, 148, 174].

The exponential growth of COVID-19 cases, especially at the beginning of 2020, put ICUs all over the world under unprecedented pressure. The drastic increase in demand for these devices exceeded the capacity of the existing supply chains, especially in regions where cross-border supply was disrupted. This created the need for a simpler, but technically suitable, machine that could be mass-produced on a very large scale and in a short time frame.

The MVM Collaboration has responded to this need by developing the Mechanical Ventilator Milano (MVM), a reliable, fail-safe, and easy to operate mechanical ventilator, built from a small number of readily available and cheap parts.

The design is inspired by the idea proposed by Manley [124] back in 1961, i.e. *the possibility of using the pressure of the gases from the anesthetic machine as the motive power for a simple apparatus to* *ventilate the lungs of the patients in the operating theater* [80], but adapting the design by replacing all moving mechanical parts with electromechanical components, allowing better parameter control and improving robustness and reliability in the long-term operation, often needed by COVID-19 patients, as also discussed in Sect. 2.2.

The MVM was designed in a collaboration between healthcare professionals and experimental physicists, benefiting from the medical expertise of the former and the latter's technical expertise in designing gas handling systems, with industrial partners (Elemaster, Italy and Vexos, Canada) who provided access to laboratories and production lines for both R&D and prototype construction, and IT researchers from academia.

The MVM was certified by the Center for Devices and Radiological Health, U.S. Food and Drug Administration (FDA) for Emergency Use Authorization in May 2020[1], in response to concerns related to the insufficient supply and availability of FDA-approved ventilators for use in healthcare settings to treat patients during the COVID-19 pandemic, and received the Health Canada Medical Device Directorate Authorization for Importation or Sale, under Interim Order for Use in Relation to COVID-19 in September 2020[2]. Moreover, it has obtained the CE marking at the beginning of May 2021, and thus it can be used and sold also in Europe. A 10,000-unit production run was recently performed in Canada (Vexos and JMP Solutions). Moreover, there is an ongoing project that aims to deliver MVM devices where they are most needed and is currently being sold by an African Union charity[3]. The cost of a single unit turned out to be about 10,000 USD, about five times less than other commercially available mechanical ventilators for ICUs.

The MVM is a mechanical ventilator for adult patients assisted with tracheal tubes, designed to directly control pressure, while the resulting delivered volume is measured. Pressure control is widely used in COVID-19 patients, who are susceptible to additional lung damage from too high pressure or volume [46, 57, 184]. The MVM can be operated in two modes, pressure-controlled ventilation (PCV), and pressure support ventilation (PSV). In PCV mode, the ventilator controls the timing of the breathing cycle and regulates the pressure applied to the patient. PCV mode is used in the acute phase of the disease when the patients are deeply sedated or paralyzed. By delivering mechanical breath with an exponentially decelerating flow pattern, PCV allows pressures to balance across lung units during a preset time, resulting in significantly reduced pressures and improved ventilation distribution. This allows lowering the risk of barotrauma attributable to the high pressures often required to ventilate

[1]`https://bit.ly/3dcZ6vs`

[2]`https://bit.ly/3OK3CfX`

[3]`https://breathoflifeafrica.org/#MentorProject`

these patients [125]. On the other hand, PSV is an assisted ventilatory mode that is patient-triggered, pressure-limited, and flow-cycled. The main use of this mode is to wean the patient off mechanical ventilation because it unloads the work of breathing and allows a gradual decrease in ventilator support until extubation [48]. Further details about the two operating modes are reported in Sect. 2.4.

Invasive mechanical ventilation exposes the patient to risks arising from infections, pneumothorax, ventilator-associated lung injury, and oxygen toxicity [137], as well as operator errors. Therefore, the MVM has a sophisticated integrated alarm system, in accordance with EN 60601-1-8:2007 [63], that monitors the various aspects of the breathing cycle and alerts the operator when any anomalies arise. In order to make it easier to control more ventilators, the MVM is directly connected to the hospital alarm system, so that the medical operators can intervene also when not near the faulty or alarmed ventilator. Hardware and software are designed to be as straightforward as possible to mitigate the risk of operator error. In addition, the MVM must be used in association with an oximeter and a capnometer. The system is designed to comply with the guidelines defined in ISO 80601-2-12:2020 [106]. The test results demonstrating compliance are discussed in Sect. 2.6.

2.2 Medical considerations on COVID-19 patients

According to current studies, approximately 5% of patients hospitalized with COVID-19 develop severe lung damage [53, 95]. This condition reflects the pathophysiology of *severe acute respiratory distress syndrome* (ARDS). ARDS is a disease characterized by reduced lung compliance due to loss of surfactant function, collapsed lung areas, and accumulation of interstitial/alveolar plasma leakage. Computerized Tomography (CT) scans demonstrate uneven distributions of aerated areas, and dense, consolidated regions of the lung; the remaining alveolar surface for gas exchange is greatly reduced in adult patients, a condition termed *baby lung* [94]. It has been suggested that the clinical management of COVID-19 patients with severe lung damage should follow the established guidelines for ARDS subjects [126, 139]. This opinion has been confirmed by a recent study comparing COVID-19 subjects to patients affected by ARDS due to other causes; the physiological differences between ARDS from COVID-19 and other causes were found to be small [95].

For this reason, the main supportive treatment for patients with ARDS is mechanical ventilation with supplemental oxygen, currently deemed the most appropriate, following a discussion that has been ongoing since the syndrome was first described in 1967 [16]. The tidal volume (V_{tidal}[4]) is a key parameter, with potentially unfavorable effects if incorrectly set, such as *ventilator-induced lung injury*. Starting in the 1970s, a V_{tidal} of $12 - 15 mL$ per kg of predicted body weight (PBW) was

[4]The tidal volume is the amount of air a person inhales during a normal breath

recommended by clinicians until, in 2000, the Acute Respiratory Distress Syndrome Network reported that the length of hospital stay and mortality could be significantly reduced using a lung-protection strategy. This strategy includes low V_{tidal} ventilation ($< 8mL$ per kg of PBW) to avoid over-distension of the baby lung, a limited plateau pressure (PP) $\leq 30cmH_2O$, and a sufficient positive end-expiratory pressure (PEEP[5]) $\leq 15cmH_2O$ [50]. PEEP targeting must be tailored to prevent lung injury due to cyclic alveolar opening and collapse and to improve oxygen delivery (amount of lung recruited) while avoiding volutrauma (lung over-distension) and cardiovascular compromise.

In addition to invasive ventilation, a series of therapies were tested in patients affected by COVID-19 pneumonia: nasal high-flow therapy, continuous positive airway pressure, and non-invasive ventilation. These strategies have been found to be suitable only in the mild and early stage of the disease, when they may be effective in stabilizing the clinical course. The early stage of COVID-19 pneumonia is characterized mainly by vascular endothelium injury, altered vasoregulation, and hypoxemia due to ventilation-perfusion mismatch. The majority of the lung is still not affected, which explains the relatively good pulmonary compliance at this stage and the interstitial edema rather than the alveolar edema seen on CT scans [126]. For patients with severe cases or who do not respond well to milder early-stage treatments, COVID-19 pneumonia may develop into ARDS, requiring treatment with an invasive ventilation device, such as the MVM. The duration of invasive ventilation treatment can last from a few days to several weeks and depends on the severity of ARDS and the presence or absence of comorbidities [81].

2.3 Design of the MVM

Fig. 2 shows a schematic of the MVM, with typical connections to the patient and to the oxygen and medical airways. In the following, the design conventions used in the figure are described. Dashed lines indicate electrical connections, and the solid lines indicate gas connections. Thick black lines represent the breathing circuit, thin red lines are connections to pressure measurements, and the green line is the gas connection to drive the pneumatic valve at the end of the expiratory line. The direction of gas flow is indicated by the blue (inspiratory phase) and red (expiratory phase) arrows. The lines in gray indicate the breathing circuit relief lines. The beige rectangle represents the main electronics and control board and the yellow square represents the supervisor board, which provides a redundant monitor and control.

The gas blender, GB-1, is external to the MVM unit. The breathing circuit and other items that get in contact with or are near the patient are replaced before each use in order to assure their sterility.

[5]The PEEP is the pressure in the lungs above atmospheric pressure that exists at the end of expiration

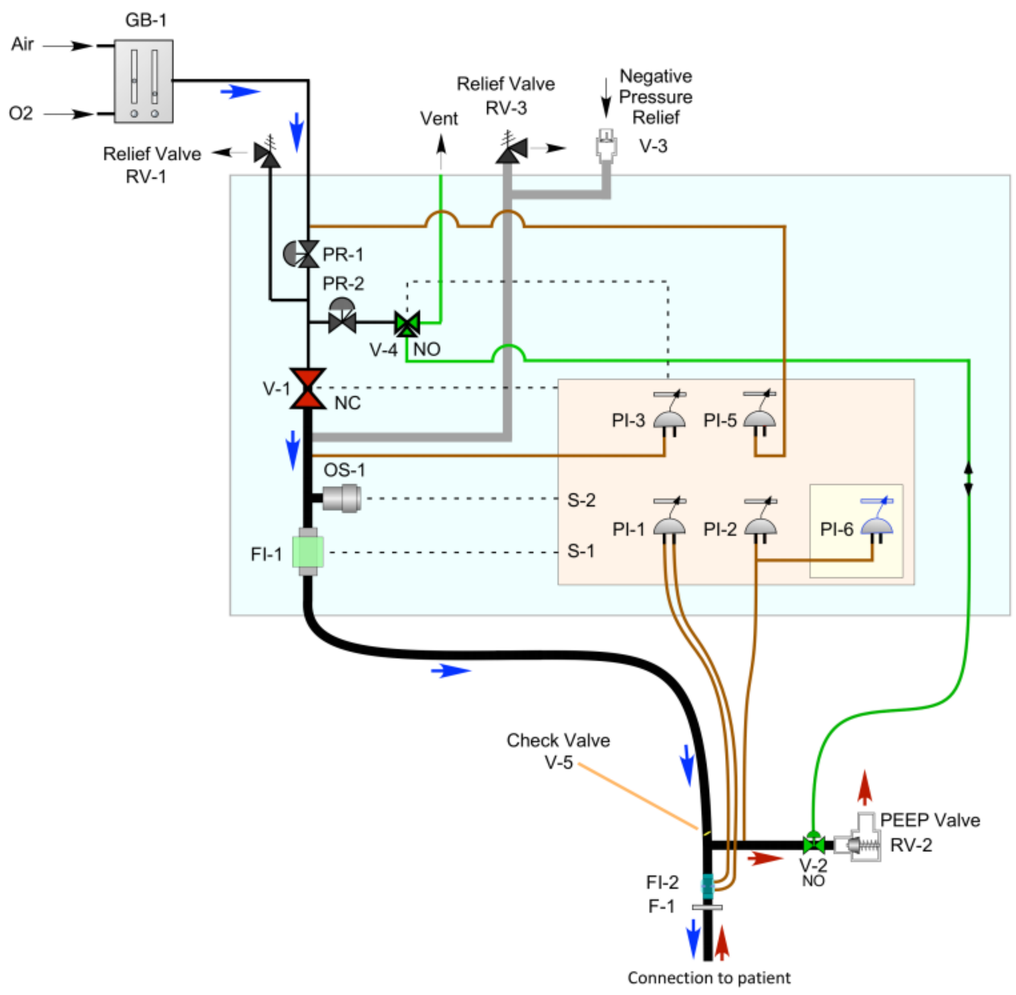


Figure 2: Schematic of the MVM ventilator system (light blue box) with the connection to the patient.

The ventilator receives its air and oxygen supply from the facility and the operator mechanically sets the fraction of inspired oxygen (FiO_2), i.e, the concentration of oxygen in the gas mixture, on the external gas-blender GB-1. The pressure out of the gas-blender is monitored by PI-5 and regulated to the appropriate pressure for the MVM by PR-1. In case of excess pressure at the MVM input, RV-1 relieves excessive air.

The MVM operates by opening the inspiratory valve, V-1, to provide the breathing gas to the patient at the desired pressure with the expiratory valve, V-2, closed. On command from the controller, V-1 is then closed and V-2 is opened to allow the patient to exhale. A mechanical PEEP valve, RV-2, sets a positive end-expiratory pressure. At the end of the expiratory phase, V-2 is closed, V-1 is opened and the cycle repeats.

V-1 is a proportional solenoid valve controlled by a loop using the pressure measured by PI-3. V-2 is a low-impedance pneumatically operated valve and is controlled by a three-way electrical valve, V-4. To avoid the need for a second source of gas, the pneumatic control is effected using the MVM input gas regulated to low pressure by PR-2. The RV-3 and V-3 relief valves prevent over-pressure

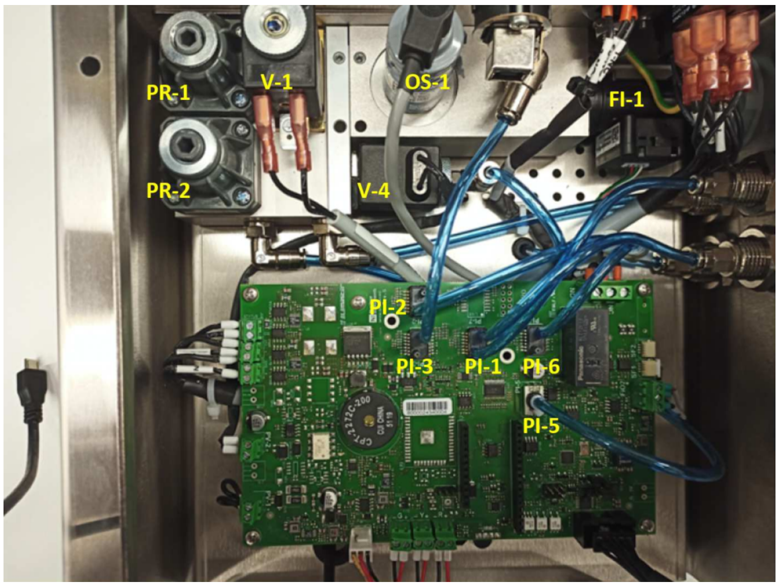


Figure 3: A view of the inside of the MVM.

and under-pressure, respectively, in the patient's line. The pressure in the expiratory line is measured by two independent indicators, PI-2 on the main control board and PI-6 on the supervisor board. The (unidirectional) flow of gas to the patient from V-1 is measured by FI-1, while the (bidirectional) flow of gas into and out of the patient is indicated by PI-1, using the pressure differential developed over FI-2. The oxygen content of the gas provided to the patient is measured with OS-1, and shown on the ventilator's graphical interface. V-5 is a check valve to prevent back-flow from the patient into the MVM.

F-1 is a bacterial filter ensuring that the air exhausted from the ventilator is free from bacteria or virus particles and thus safe for doctors and nurses surrounding the patient. Indeed, this filter prevents the large diffusion of droplets that would otherwise be emitted in the surroundings, as demonstrated by many flow visualization studies [67, 68, 138], and could spread COVID-19 (or other viruses).

Fig. 3 shows the inside of the stainless steel enclosure of the MVM, with the pneumatic control components and the electronics board. Furthermore, the yellow labels on the electronic components identify them, as in Fig. 2.

2.4 MVM operating modes

As anticipated in Sect. 2.1, the MVM is designed to ventilate patients in two different modes, namely *Pressure-Controlled Ventilation* (PCV) and *Pressure-Support Ventilation* (PSV). The basic concepts of the two ventilation strategies are analyzed below.

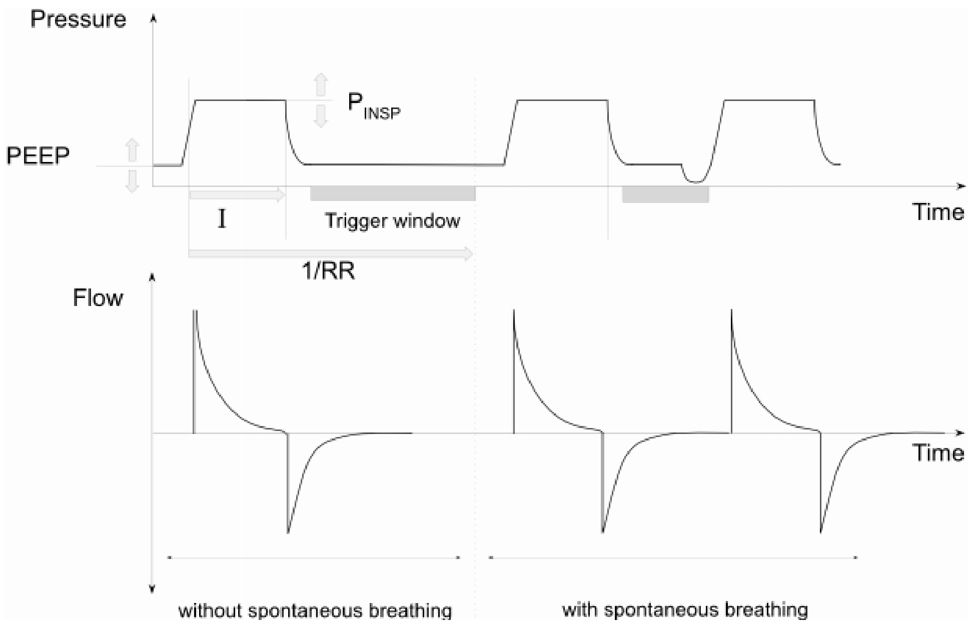


Figure 4: Respiratory cycle during PCV ventilation.

2.4.1 Pressure-Controlled Ventilation Mode

PCV is a time-cycled ventilation mode in which the operator sets the inspiratory pressure, the PEEP, the duration of the inspiratory phase of the breathing cycle, and the number of breaths per minute. As flow and volume are not directly set, the resultant patient tidal volume varies, depending on lung compliance and resistance, patient effort, and inspiratory pressure.

Fig. 4 shows the respiratory cycle during PCV ventilation. A new inspiration begins either after a breathing cycle is completed according to the set *respiratory rate* (RR), or if the MVM detects the initiation of a breath by the patient before the cycle is complete and the inhale trigger criteria are met. The trigger window for a patient-initiated breath occurs during the expiratory phase of the previous breath. When inspiration begins, the MVM provides the patient with the set inspiratory pressure (P_{insp}) for the set duration of the inspiratory phase of the breathing cycle. The respiratory rate and the ratio between the inspiratory and expiratory times (I:E) are the parameters that control the time cycle.

2.4.2 Pressure-Support Ventilation Mode

In PSV mode, the MVM provides pressure to help the patient breathe, while the patient controls the RR. This mode is not suitable for patients who cannot initiate breathing on their own. A pressure-support breath is initiated when the MVM detects a sudden pressure drop, which indicates the start of patient inspiration. Such sudden drops in the pressure are determined by measuring the changes in the rate at which the pressure is decreasing, indicated by the downward curvature near the start of the pressure vs. time waveform.

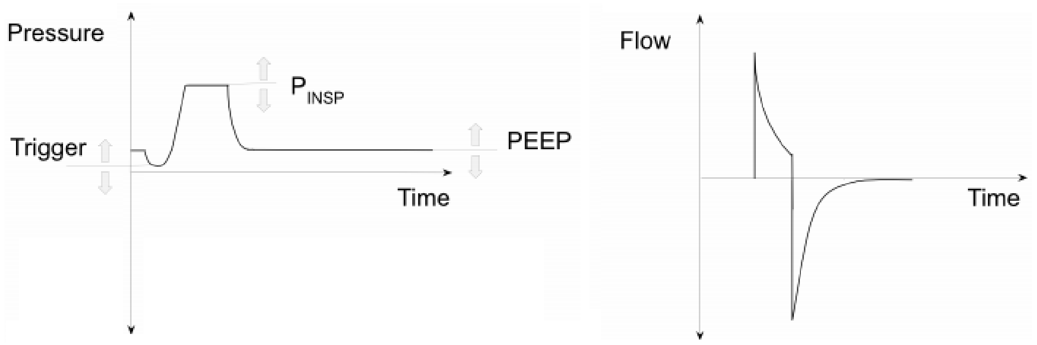


Figure 5: Respiratory cycle during PSV ventilation.

Fig. 5 shows the respiratory cycle during PSV ventilation. When a pressure-support breath is triggered, the MVM increases the pressure to the set P_{insp}. Then, when the patient's inspiratory flow drops below 30% of its peak, the MVM ends inspiration and returns pressure to the baseline, allowing exhalation. If the patient does not trigger a breath within a set apnea-trigger time window, the MVM switches to PCV mode, and an apnea alarm, which must be reset by the operator, is activated.

2.5 MVM electronics and software

Electronics and software are responsible for controlling valve systems, reading ventilation parameters (e.g., pressure, flow, and oxygen concentration), generating audible and visual alarms in hardware (including LEDs, and buzzers), monitoring the correct ventilation, and interacting with the operator. Both electronics and software are composed of three main macro-components:

- *Graphical user interface* (GUI), a touch-screen panel that displays the information needed to check the respiratory condition, allows parameter setting, and displays ventilation parameters and alarm settings.
- *Controller*, which receives operator input from the GUI, communicates with the valve controllers, serial interfaces, and other sub-components, and sends to them commands.
- *Supervisor*, that monitors the overall behavior of the system and ensures that the machine runs safely.

2.5.1 Hardware

The MVM operations are managed by an electronic board hosting all the components required to measure the relevant quantities, drive input and output valves, and activate visual and audio signals for the operator. The board houses a microcontroller (ESP32), a Raspberry Pi 4, and the supervisor.

The ESP32 includes a dual-core 240 MHz microcontroller, 0.5 MB of RAM, Wi-Fi, and Bluetooth connectivity that are deactivated, in order to comply with the regulations on medical devices connectivity[6]. The ESP32-based solution is widely used in the IoT environment, hence it is readily available, and is programmed using an Arduino core. A USB connection between the ESP32 and the Raspberry Pi enables the transmission of commands and settings from the GUI to the controller and the read-back of the system status.

The supervisor board includes a microcontroller that is programmed with the standard Arduino bootloader to allow firmware updates via an optoisolated serial connection with the Raspberry Pi. This connection is also used to enable monitoring during the ventilation, in order to check the safety of each ventilation state.

The power is provided by an external unit, equipped with a battery ensuring 2 hours of autonomy, that generates two independent 12 V sources. One is regulated with step-down converters to 3.3 V and 5 V, as required to operate the sensors and the ESP32. The other one provides power to the valves, to the supervisor, and to the Raspberry Pi. In this way, a failure of either of the supply lines would still leave a microcontroller active to alert the operator and to return to a safe state.

Three I2C buses connect the sensors and the microcontrollers. The main I2C bus connects the pressure sensors PI-1 and PI-2, and the flow-meter FI-1 to the ESP32. Two ADCs connect to the main bus, digitizing the readings from the FiO_2 analog oxygen sensor and the PI-5 analog pressure sensor, and monitoring internal voltages. The main I2C bus connects the supervisor to the controller, enabling the watchdog function. A priority bus connects the PI-3 sensor to the ESP32 and allows it to be polled at frequencies over 1 kHz, as required by the fast proportional–integral–derivative pressure controller, used for handling the input valve. A third dedicated I2C bus connects the supervisor to the PI-6 sensor and an ADC that monitors the board's internal voltages. This auxiliary bus ensures normal supervisor operation in case the main I2C bus freezes.

The control boards include ON/OFF valve controls, current-feedback valve controllers, and visual and audio alarm circuits.

2.5.2 Software

The high-level software architecture, shown in Fig. 6, illustrates the communication among the three software components, namely the controller, the supervisor, and the GUI.

[6]For safety reasons, not all the certification standards allow using wireless connections to control medical devices.

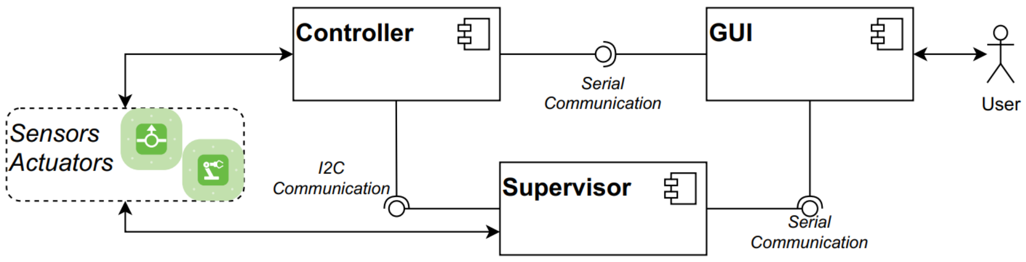


Figure 6: The high-level MVM software architecture.

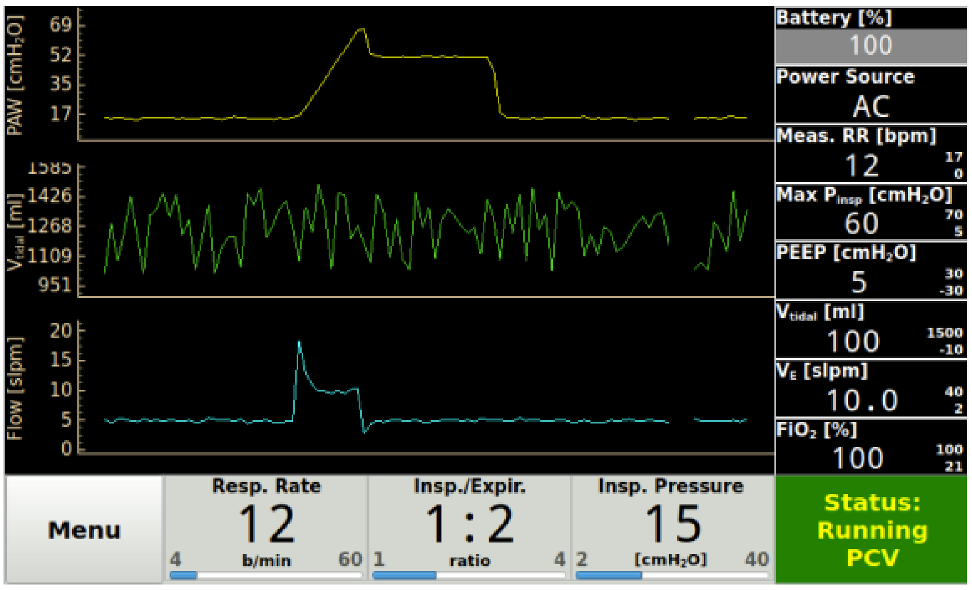


Figure 7: The MVM GUI.

GUI

When the MVM is turned on, the graphical user interface guides the operator through startup procedures, including setting operating parameters and alarm thresholds and performing hardware and software self-tests, aiming at checking that every component of the MVM works as expected.
The GUI home screen is divided into three parts, as shown in Fig. 7. The central part displays the three monitored parameter waveforms (airway pressure, inspiratory tidal volume, and airflow). A side panel displays the current value of other monitored parameters, alarms, and warnings, while the bottom part is dedicated to parameter setting and to the menu buttons. The GUI is written in Python3, using the PyQt5 library, and runs on a Raspberry Pi, chosen for its wide availability and its computing-power to power-consumption ratio.

Controller

The controller software, implemented in C++, receives inputs from the GUI and interacts directly with the hardware, by receiving patient breathing data and issuing commands, and vice-versa. Its software is divided into four sub-modules:

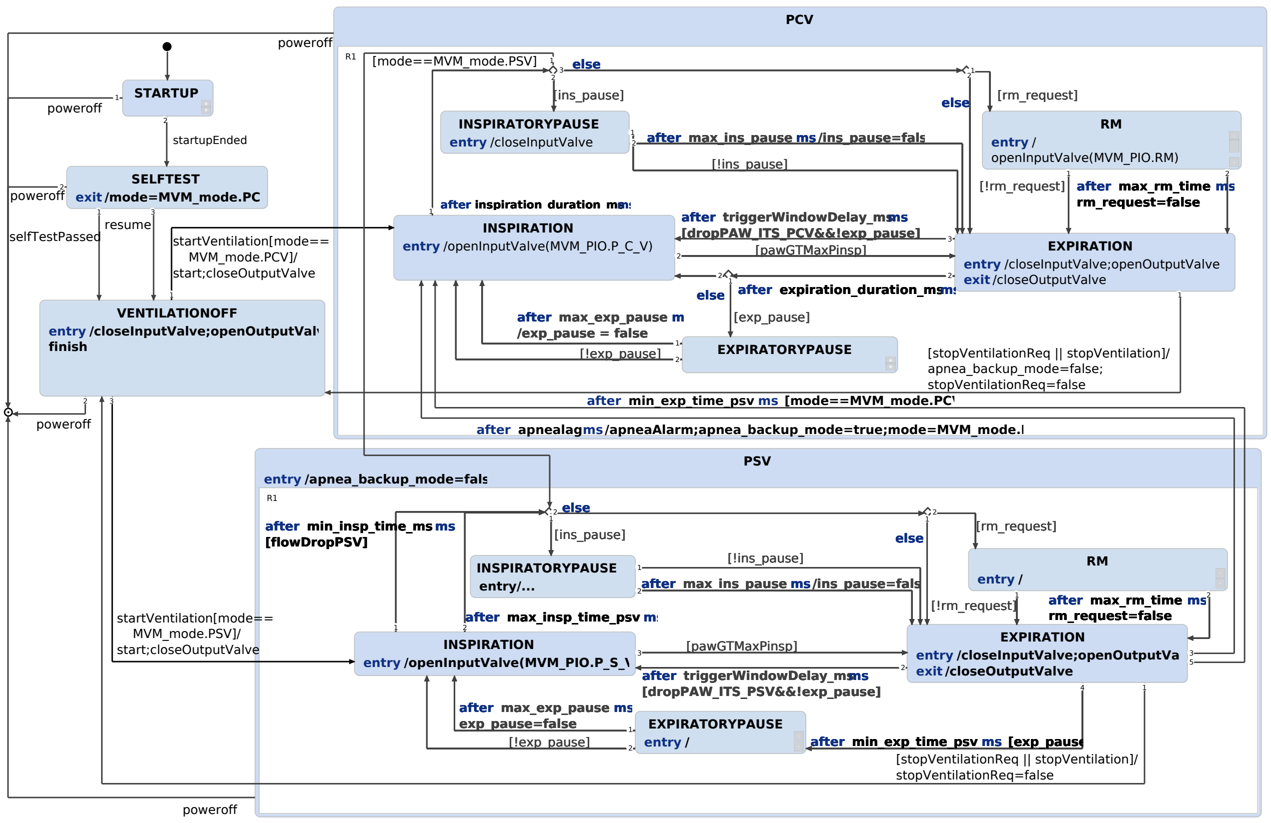

PCV
poweroff
STARTUP
startupEnded
SELFTEST
exit /mode=MVM_mode.PC
resume
selfTestPassed
startVentilation[mode==MVM_mode.PCV]/start;closeOutputValve
VENTILATIONOFF
entry /closeInputValve;openOutputVal
finish
[mode==MVM_mode.PSV]
else
[ins_pause]
INSPIRATORYPAUSE
entry /closeInputValve
after max_ins_pause ms/ins_pause=fals
[!ins_pause]
[rm_request]
RM
entry / openInputValve(MVM_PIO.RM)
[!rm_request]
after max_rm_time ms rm_request=false
after inspiration_duration ms
INSPIRATION
entry /openInputValve(MVM_PIO.P_C_V)
after triggerWindowDelay_ms ms [dropPAW_ITS_PCV&&!exp_pause]
[pawGTMaxPinsp]
EXPIRATION
entry /closeInputValve;openOutputValve
exit /closeOutputValve
after expiration_duration_ms ms
[exp_pause]
after max_exp_pause ms /exp_pause = false
[!exp_pause]
EXPIRATORYPAUSE
[stopVentilationReq || stopVentilation]/ apnea_backup_mode=false; stopVentilationReq=false
after min_exp_time_psv ms [mode==MVM_mode.PC
after apnealag ms/apneaAlarm;apnea_backup_mode=true;mode=MVM_mode.
PSV
entry /apnea_backup_mode=fals
after min_insp_time_ms ms [flowDropPSV]
else
[ins_pause]
INSPIRATORYPAUSE
entry/...
[!ins_pause]
after max_ins_pause ms/ins_pause=fals
[rm_request]
RM
entry /
[!rm_request]
after max_rm_time ms rm_request=false
after max_insp_time_psv ms
startVentilation[mode==MVM_mode.PSV]/ start;closeOutputValve
INSPIRATION
entry /openInputValve(MVM_PIO.P_S_V
[pawGTMaxPinsp]
EXPIRATION
entry /closeInputValve;openOutputVa
exit /closeOutputValve
after triggerWindowDelay_ms ms [dropPAW_ITS_PSV&&!exp_pause]
after max_exp_pause ms exp_pause=false
[!exp_pause]
EXPIRATORYPAUSE
entry /
after min_exp_time_psv ms [exp_pause
[stopVentilationReq || stopVentilation]/ stopVentilationReq=false
poweroff

Figure 8: The Yakindu State Machine for the MVM controller.

- the *interface* allows the communication with the GUI and the supervisor;
- the *monitor* observes the sensors and the status of the system and triggers alarms;
- the *control* changes the respiratory phases using a state machine and controls valves;
- the *hardware drivers* open and close valves and raise visual and audio alarms.

The operation modes are implemented in the controller with a state machine. Fig. 8 shows the state machine modeled with the Yakindu Statechart Tool (Itemis AG). It interacts with the valve controller, opens and closes the expiratory valve V-2, and sets the desired pressure for the inspiratory valve V-1. The corresponding C++ code has been generated from the state-chart model and integrated into the controller logic.

The state machine describes the states in which the device can operate and the transitions between them. In particular, the machine starts in the `StartUp` state and needs to complete all the `SelfTests` before operating, either in PCV or PSV mode. The `SelfTests` take about ten minutes to complete, assuming the doctor already knows the parameters to be set for the alarm thresholds and has been using the MVM before.

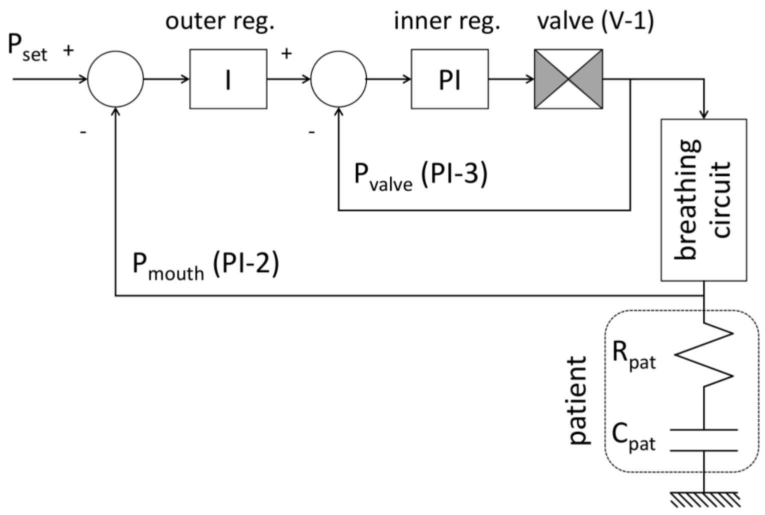


Figure 9: Schematics of the pressure controller of the MVM.

During the breathing cycle, the MVM controls the valve V-1 to increase the pressure during the inspiration phase for the prescribed time duration. The actual tidal volume depends on the patient's response and on how quickly the regulator reaches the pressure set point, P_{set}. To protect the patient, the device must never overshoot this point significantly, by controlling the pressure peaks. To satisfy these objectives and constraints, the control algorithm has been designed to reach P_{set} in a fixed rise time, and without overshooting. For this reason, the MVM implements a control architecture composed of two nested control loops, often used in safety-critical industrial products, that allows a simpler controller tuning, based on first-principle considerations, and ensures sufficient robustness against disturbances.

Fig. 9 shows the controller structure and the simplified parameters used to model the patient's lungs, namely *linear resistance* R_{pat}, and compliance C_{pat}. The outer loop regulates the pressure at the patient by reading the value given by the PI-2 sensor, read at 300 Hz rate; it is controlled by a simple integrator, whose time constant depends only on the desired rise time. On the other hand, the inner loop controller is a proportional-integral regulator that is automatically tuned through patient parameters (R_{pat} and C_{pat}) and is fed back with the pressure at the valve outlet PI-3, read at 1 kHz rate. To be adaptable to every patient, patient parameters are automatically determined during the first three respiratory cycles. To ensure patient safety during this phase, the inner controller parameters are fixed and set to ensure smooth pressure rise in 300-500 ms, over the expected range of R_{pat} and C_{pat}, with an overshoot always lower than 2 mbar. A recursive least-squares method is used to estimate patient parameters, limiting memory use in the microcontroller. Then, the identified R_{pat} and C_{pat} are used to tune the inner loop and the time constant of the outer loop to reach a more desirable rise time of about

100 ms. Note that model parameters, estimated after three respiratory cycles, can be continuously updated and re-tuned as necessary, upon the doctor's request.

Supervisor

The supervisor software is implemented in C++ and is responsible for monitoring the controller, the GUI, and the hardware. In case of errors, it raises alarms if not already raised by the controller or the GUI, ensuring patient safety. For instance, if the pressure in the circuit exceeds the maximum allowed value for a given duration, the supervisor switches off the ventilation and brings the valves into the safe position (valve V-1 closed and V-2 open), which allows the patient to breathe.
Like the controller, the supervisor has an integrated state machine that models its behavior. After startup, it waits for the operator to start the self-test procedure. Then, the supervisor alternates between two operation modes, namely `breathing off` (the MVM is ready to work but it is not ventilating) and `breathing on` (the device is ventilating). To ensure the patient's safety, the supervisor can move into a fail-safe state from any state, in case of errors.

Software certification

As already introduced in Sect. 1.2, every medical device must comply with the IEC 62304:2015 regulation. More details about the process followed, some of the lessons learned from the certification effort, and guidelines will be reported in Chapter 3.

2.6 Device testing

Taking into account the hardware and software of the MVM as a whole, all standards require a system testing activity, using different ventilation modes. During this activity, failures are simulated, and the long-term durability of the device is demonstrated.

2.6.1 Software testing

In order to achieve the software certification and to assure the quality and reliability of the MVM, a regular software testing process has been carried out together with the software development. Indeed, as I will later present in details in Chapter 3, during the development of the MVM we have followed an agile-like process, in which software development and software testing, for each software unit, have been performed in parallel. Given that the most of the participants to the development of the MVM were not software testers, no particular methodologies (e.g., model-based testing, combinatorial testing,

Parameter	Range	Units
PP	$\pm(2 + (4\% \ of \ set \ value))$	cmH_2O
$PEEP$	$\pm(2 + (4\% \ of \ set \ BaselineAirwayPressure \ value))$	cmH_2O
V_{tidal}	$\pm(4 + (15\% \ of \ measured \ value))$	mL
RR	$\pm(0.5 + (5\% \ of \ set \ value))$	min^{-1}
$I : E$	$\pm(0.1 + (5\% \ of \ set \ value))$	

Table 2: Acceptable ranges for the measured breathing parameters of interest.

etc.) have been adopted, but only simple test cases, written by hands and guided by the code coverage have been used. This complex process lasted for more than two months, considering altogether the testing activities.

2.6.2 Tests in PCV mode

The test setup that have been used in this phase is equivalent to that described in Fig. 201.102 in the ISO 80601-2-12:2020 reference standard [106]. In particular, the breathing simulator (*IngMar Medical - ASL 5000*) is used both as a test lung (with settable compliance and resistance) and as an independent sensor for pressure, flux, temperature, and oxygen concentration.

This series of tests refers to the ISO reference standard, Section 201.12 for pressure-controlled inflation-type testing, Subsection 201.12.1.102, verifying that the values of the breathing parameters set and measured by the MVM agree within the declared accuracy range. The only exception is the value of tidal volume V_{tidal}, which is not set in MVM, for which the measured value is compared with the value independently measured by the breathing simulator.

In Tab. 2 the acceptable ranges for the measured breathing parameters of interest are reported. Out of the 21 tests in the ISO standard Table 201.105, only the first 11 have been performed, as they are the ones involving tidal volumes in the range relevant for MVM operation ($50mL \leq V_{tidal} \leq 500mL$). For each condition, the testers ensured that the breaths resulted in smooth and reproducible time traces, such as the one shown in Fig. 10. Measurements were taken over 30 cycles under steady state conditions. The plateau pressure (PP) and PEEP were measured in the last $50ms$ of the inspiratory and expiratory phases, respectively, while V_{tidal} was obtained by averaging the MVM measurement over 30 cycles and compared to breathing simulator measurements, averaged over the same cycles. RR and I:E were calculated from the collected waveforms using custom algorithms, which has allowed testers to verify that the calculated values agree well with those from the breathing simulator. When comparing the breathing simulator measurements to the values set in MVM (or in the case of V_{tidal}, to

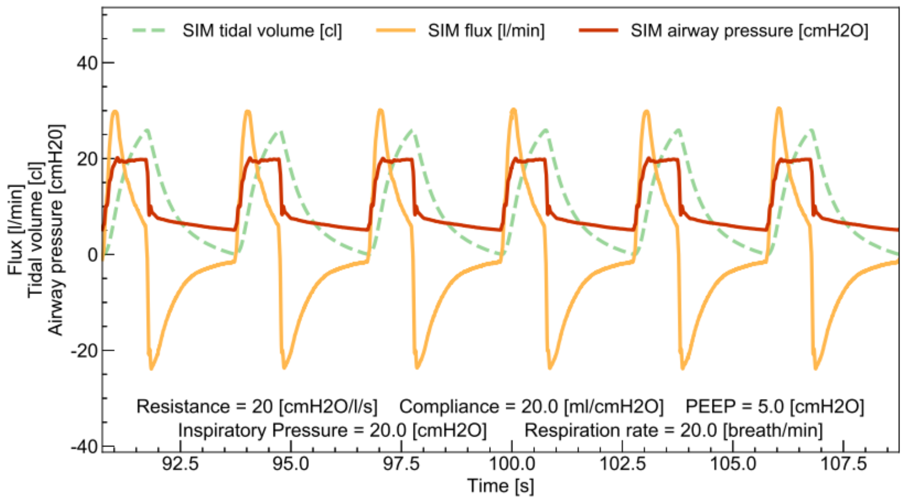


Figure 10: Example waveforms from the breathing simulator with the MVM in PCV mode.

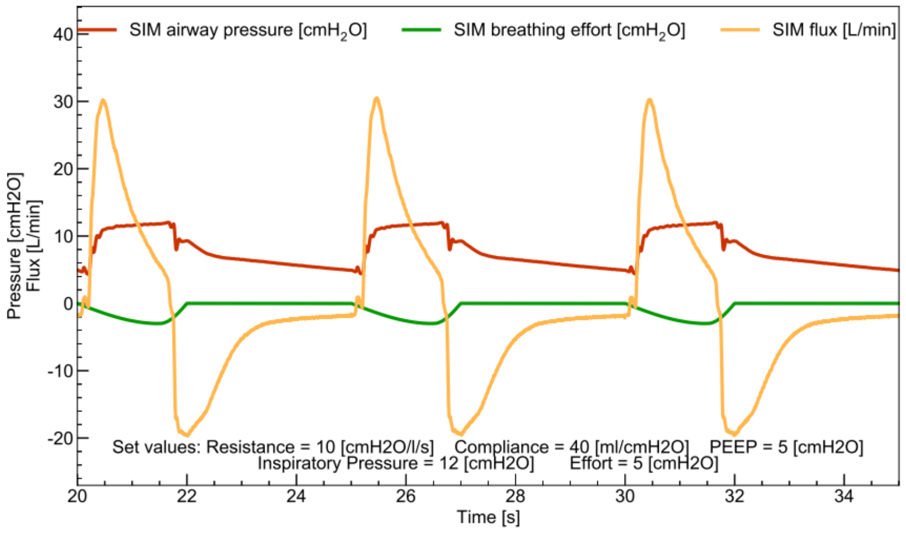


Figure 11: Example waveforms from the breathing simulator with the MVM in PSV mode.

the MVM reported value), all breathing parameters have been found to be within the tolerances given in Tab. 2, for all 30 cycles.

2.6.3 Tests in PSV mode

In PSV mode, the patient actively initiates a breathing cycle by producing a decrease in airway pressure. Then, the ventilator must readily recognize this decrease and provide airflow support in a way not to stress the patient's compromised respiratory system. The MVM must also recognize the patient-driven end of the inspiratory phase and begin the expiratory phase. To achieve a quick recognition of the patient breathing effort, a trigger system based on the second derivative of the airway pressure with respect to time was devised. Its sensitivity is set by varying the threshold on the maximum value of this parameter.

A typical waveform of the MVM operating in PSV mode is shown in Fig. 11. Thanks to the sensitivity of the trigger, even in the case of a patient's breathing effort as low as $2cmH_2O$, the MVM is able to recognize the effort within $100 - 200ms$ and starts the inspiration by providing the desired pressure support.

2.6.4 Single-fault test condition based on ISO protocols

A medical device, regardless of its software, must be robust even w.r.t. specific single faults conditions (see Section 201.13.2.101 of the ISO standard [106]). The tested conditions relate to the disruption, disconnection, or bad connection of the external components, such as the gas delivery to the patient pathway or the pressure to the patient sensors. The tests performed ensure that when any one of the fault conditions is triggered, the system successfully maintains the patient's breathing parameters in the safety zone and triggers an alarm. For this kind of testing, the fault conditions are induced manually by disconnecting external components during the test operation, such as the PI-2 sensor. Once the default conditions are restored, the ventilator performance is automatically re-established.

2.6.5 Long-term durability tests

As required by the standards, several units were continuously tested for a period of three months. During this testing, the units met all the criteria defined for correct operation, as no alarms of any kind were recorded.

2.6.6 Response of the MVM to an increased oxygen concentration

Paragraph 201.12.1.105 of the ISO standard [106] requires evaluating the ventilator's speed of response to a change in the FiO_2 set point, i.e., the oxygen concentration. This test involves measuring t_{rise} (i.e., the time required for the oxygen concentration in the lung to rise from 21% to 90% when the input FiO_2 is suddenly increased to 100%. This test has been performed by connecting the MVM to a gas analyzer, which measures oxygen concentration, and to an adjustable test lung, setting the parameters as in Tab. 3 in the MVM, and starting it in PCV mode with FiO_2 at 21%. Once steady-state conditions are reached, FiO_2 is then suddenly increased to 100%. The measured time t_{rise} for the oxygen concentration to reach 90% during the expiratory phase is reported in Table 3.

2.6.7 Evaluation of bio-compatibility

Bio-compatibility of breathing gas pathways of a brand-new MVM was assessed according to the ISO 18562-3:2017 guideline [103]. The analyses were carried out using a thermal desorption unit coupled

V_{tidal} $[mL]$	$I : E$	RR $[min^{-1}]$	R_{pat} $[cmH_2O/(Ls)]$	C_{pat} $[mL/cmH_2O]$	t_{rise} $[s]$
500	1:2	10	5	20	76
150	1:2	20	20	10	85

Table 3: Parameters used to test the MVM response to increase in FiO_2.

to gas-chromatography and mass spectrometry detection, in accordance with the ISO 16000-6:2011 guideline [100]. The tests showed that volatile emissions from a brand-new MVM system are limited to a few chemicals, which mainly belong to the siloxanes family. The presence of such chemicals was somehow expected, due to the large use of silicones in biomedical devices. However, all values complied with the permissible levels suggested by ISO 18562-3:2017 [103] and decreased after one day of use as they were washed away from medical air.

2.7 Conclusion

The Mechanical Ventilator Milano, a novel intensive therapy mechanical ventilator designed for rapid, large-scale, low-cost production for the COVID-19 pandemic, has been conceived, designed, prototyped, and tested by a unique international collaboration of scientists, medical specialists, and industrial partners. Due to its complexity, composition of both hardware and software (structured as a set of several modules) and the successful process of device certification, it is a complete example of a medical device, which quality has to be assured. Thus, in the next chapters, I will often use it as a case study for conducting further experiments.

Chapter 3. The software certification process: lessons learned and guidelines

In this chapter, lessons learned during the development of the Mechanical Ventilator Milano (MVM) are presented. Moreover, through validation activities, some empirical guidelines for the development of medical devices are derived starting from the lessons learned. These guidelines are proposed to be used under emergencies, but some of them can be generalized to every software development condition. This chapter is based on [22, 24] and is structured as follows. Sect. 3.1 introduces the work presented in this chapter, by highlighting the objective and the research questions that have been used for the definition of the lessons learned and guidelines. Sect. 3.2 presents the research methodology used by me and the research team I was part of for the identification of the lessons learned, their validation and the derivation of guidelines. In Sect. 3.3 and Sect. 3.4 the lessons learned and guidelines that have been identified are respectively detailed and validated. Finally, Sect. 3.5 concludes the chapter.

3.1 Introduction

The development of safety-critical devices, especially when they are used in the medical domain, considers certification as a mandatory step [82, 85, 112] since a software failure or malfunction can compromise the health of human beings that interact with it. In particular, as reported in Sect. 1.2, in healthcare, devices must comply with the IEC 62304 [64] standard and the FDA guidelines [83].
The importance of a well-documented and correct software life cycle in the development of medical systems has been discussed in [158] and [116]. Literature confirms that applying a well-documented life cycle is not limited to just assuring the safety and the certification of medical devices [176], since adopting frameworks and the good principles of software engineering can help developers to compare different medical devices for identifying gaps between them and improve the capabilities of products [140]. However, one may argue that using a well-documented and correct life cycle when developing medical software under emergency (such as for the MVM case study during the COVID-19 pandemic – see Chapter 2) is a time-consuming activity that cannot be fully followed. In fact, the focus on well-documented and regulated frameworks poses some constraints on the adoption of agile software development techniques or requires the adaptation of selected agile methods and practices [131]. Nevertheless, in the literature, we can find attempts to apply agile practices also in medical software development planning, for example, by integrating the more classical V-model with agile methodologies [129, 132].

In this chapter, I present the experience acquired during the work on the mechanical ventilator and some of the lessons learned that aim to speed up the development process while still satisfying the safety standards, even under emergency. By emergency, I mean producing software under the following constraints: 1. the first hard constraint is time, meaning that the software device should be produced as soon as possible; 2. the second hard constraint concerns establishing a development team in a hurry, in an emergent and voluntary fashion, based on the personal network, heterogeneous under various dimensions, and composed of people that dedicate their private time to the project, while still continuing their normal job. In other emergency situations, like hurricanes or earthquakes, there can be additional constraints like lack of energy power or Internet connection, but this chapter only presents lessons learned and guidelines derived from the MVM experience, and thus I limit myself to the limitations observed during the MVM project.

In particular, this chapter tries to answer the following research question: "Are there any empirical guidelines for developing medical software under emergency?". More specifically, it can be detailed in the following three aspects:

- Which development process is most appropriate for the development of safety-critical devices under emergency?
- How can the activities of the development process be performed in order to be simplified and sped up under emergency, while still maintaining the rigorousness required by certification standards?
- How to deal with a heterogeneous development team built in an emergent and voluntary fashion?

3.2 Research methodology

In this section, I present the research methodology followed during the work presented in this chapter. It is based on a case-study approach [155] since it is considered the most appropriate research methodology to study a phenomenon in its natural context, i.e., when the phenomenon is difficult to study in isolation. Indeed, in the MVM project, it is difficult to clearly and precisely identify and delimit in a real context the process and the activities to be followed when producing software devices that are supposed to be compliant with safety standards under emergency.

Fig. 12 presents an overview of the research methodology. In the following, the three main steps of the researches presented in this chapter are explained in details.

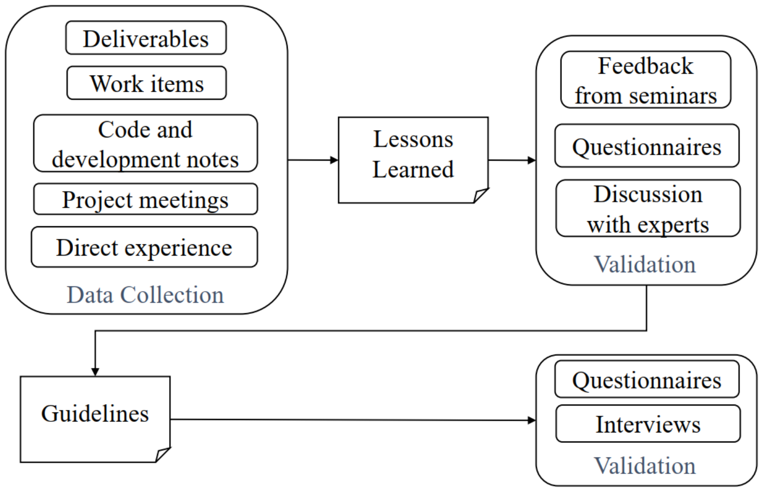


Figure 12: Research Methodology.

Activity	# People	Deliverables
System development plan	3	5
Supporting activities	22	12
System requirements	5	1
Software Architecture Design & Risk Management	10	3
Software Requirement Spec.	21	15
Software Detailed Design Impl.	18	N/A
Unit Testing	22	20
Integration Testing	11	2
Validation Testing	9	2

Table 4: Summary of the effort required for each phase.

3.2.1 Data collection and analysis

For identifying lessons learned and guidelines, data has been collected along the various activities of the software development process, namely during requirements engineering, architectural design, testing (unit, integration, and validation), implementation, documentation, and traceability checking. To provide an overview of the project activities that created the data for this chapter, Table 4 summarizes the effort required in terms of (i) activities performed; (ii) number of people involved in each activity; and (iii) deliverables produced for documentation – each of these is a Microsoft Word document.

Collected data were heterogeneous: they consist of the various deliverables created for the certification purpose and stored in a Google Drive shared – among the members of the certification team – folder, but also of other items created during the project completion, such as whiteboard sketches, notes (personal or shared within sub-teams), source code, comments in the code, changes requests, test reports, emails, and so on.

All this data allowed to formulate a set of lessons learned, which are reported in Sect. 3.3, and to derive from them a set of guidelines, reported in Sect. 3.4.

3.2.2 Validation methodology for the lessons learned

Sect. 3.3.3 presents the validation of the lessons learned. They have been discussed in three international events, where I and the development team members have been invited to provide keynotes or invited presentations. Moreover, they have been validated through lectures for PhD courses and seminars provided in three European universities. Finally, the largest validation has been executed in the context of two courses for the Italian engineering society, held in date September 20th 2021 and October 2nd 2021 as virtual events, which had, overall, 56 attendees. During these courses, participants have responded to questionnaires aimed at checking the level of agreement and importance of each lesson learned (see the replication package [23]), by alternating sessions in which they were provided with new content with validation sessions in which the questionnaires were filled. In total, 56 complete answers have been analyzed to draw conclusions.
The data collected during the validation of the lessons learned enabled to formulate a set of guidelines that may help during the development of safety-critical systems under emergency. In Sect. 3.4, I present the guidelines and show how they have been synthesized from the lessons learned.

3.2.3 Validation methodology for the guidelines

The guidelines have been validated via questionnaires (see the replication package [23]) and interviews. In particular, validation activities involved experts in the development of safety-critical systems in healthcare, but also in other domains. Since the validation process aimed at identifying patterns in the replies, it has been done through identical copies of the questionnaires that have been distributed to various groups of experts (i.e., (*i*) experts in health care safety-critical systems, (*ii*) experts in safety-critical systems in other domains, mostly automotive, (*iii*) experts in agile and safety-critical systems, (*iv*) experts involved in the development of the MVM, (*v*) experts in computer science, and, finally, (*vi*) developers of other ventilators under emergency) to grasp different points of view.
Table 5 provides an overview of the experts involved in the validation. In total, the validation effort involved 37 experts, which are practitioners or academics, answering the questionnaires (21 practitioners and 16 academics), and 9 of them accepted to be contacted for a video call interview (7 practitioners and 2 academics) to better understand their responses and feedbacks. Those experts that performed an interview are highlighted by a * symbol in Table 5.
Our questionnaires mostly contained closed-ended questions, except for an optional open-ended question at the end of the two groups of questions which was inserted to collect free comments from the participants. The interviews were driven by the content of the questionnaires and their responses. For each interview, the main criticalities or stronger agreements of the specific expert were then discussed

ID	Years	Industry/Academia/ Other	Role
1	< 1	Open Source Ventilator Team	
2	1-3	Industry	Mechanical engineer
3*	4-5	Industry	CTO
4	>20	Academia	Director of Technology
5	<1	Academia	Scientist
6	11-20	Academia	Assistant professor
7	<1	Academia	PhD student
8*	<1	Academia	Assistant professor
9	4-5	Academia	Associate professor
10	<1	Academia	PostDoc
11	4-5	Academia	PostDoc
12	<1	Academia	PostDoc
13	<1	Academia	Associate professor
14	>20	Industry	CTO
15	<1	Academia	Associate professor
16*	>20	Industry	Program manager/Quality manager
17	1-3	Industry	Developer
18	6-10	Industry	System/Software architect
19	<1	Industry	Developer
20	6-10	Industry	Developer
21	11-20	Industry	System/Software architect
22	1-3	Academia	Assistant professor
23*	4-5	Academia	Associate professor
24	1-3	Academia	Assistant professor
25	6-10	Industry	Developer
26	6-10	Industry	System manager
27	11-20	Industry	Developer
28	1-3	Industry	CTO
29	11-20	Industry	CTO
30*	1-3	Industry	Developer
31	11-20	Academia	Full professor
32*	11-20	Industry	System manager
33*	>20	High risk systems	
34*	>20	High risk systems	
35	1-3	Academia	Assistant professor
36	4-5	Industry	Technical expert of software quality
37*	6-10	Industry	System/Software architect

Table 5: Experts overview. The column year stands for "Years of experience in the development of critical software". The IDs with asterisk are experts interviewed.

in depth. The interviews have been performed by a minimum of 4 co-authors: one co-author had the role of the driver of the interview, supported by another and the other two co-authors were mainly responsible for taking notes. At the end of each interview, the co-authors met to summarize the outcomes and findings of the interview.

By analyzing the validation results on each guideline, here I discuss the benefits and risks of each of them and clarify their scope (see Sect. 3.4.3).

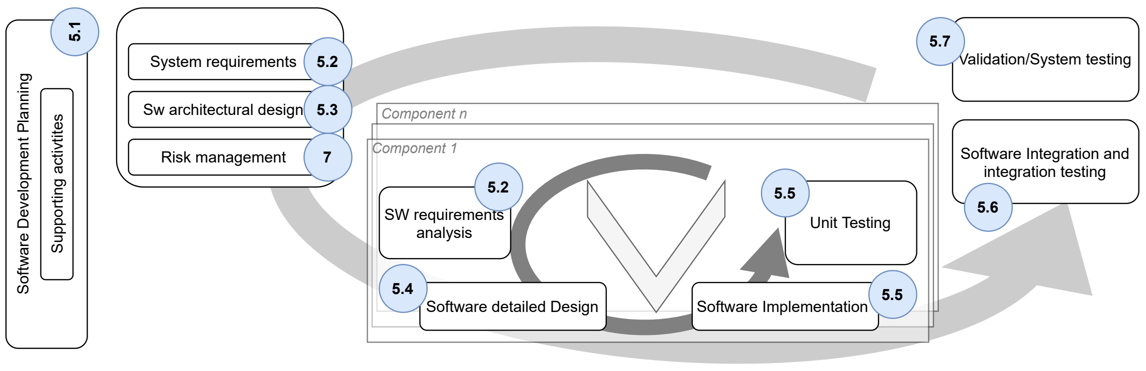


Figure 13: MVM software development process.

3.3 Lessons learned

As presented in Sect. 3.1, during the development of the MVM, several actions and activities have been carried out to ensure the quality of its software according to the IEC 62304 standard [64]. In this section, I present the process I and the research team I was part of followed, all the performed activities with their relevant details, and the lessons learned during them. Some of the activities are useful for the whole development process (e.g., planning activities and tools definition), while others are more connected to a specific development phase (e.g., requirement analysis or software design).

3.3.1 Development process

As required by the standards and regulations for medical software (see Section 1.2 for more details), the first activity to be performed while developing medical software is the definition of the software development process, together with all supporting activities aimed at defining processes, task responsibilities, and tools to be used.

Software development planning

First, the device has been classified based on its potential to cause injuries, as required by the IEC 62304 standard. Starting from the safety classification and taking into account the activities required by the IEC 62304 standard (which are summarized in Sect. 1.2.1 and in Fig. 1), the MVM as a whole has been classified in class C since death or serious injury is possible. Considering the safety class and the mandatory activities, the team has defined the development process to be adopted, depicted in Fig. 13, which integrates the classical V-model with agile practices, aiming at improving the rapidity and flexibility of software development, while still guaranteeing the expected safety of the device.
The list of activities included in the adopted software development process is as follows:

1. *Software Development Planning*, which regards the entire MVM and maps to activity 5.1 in the standard, including all supporting activities;

2. *System requirements* and *Software architectural design*, which refer to the MVM as a whole and define the desired components, after a *risk analysis* has been performed;

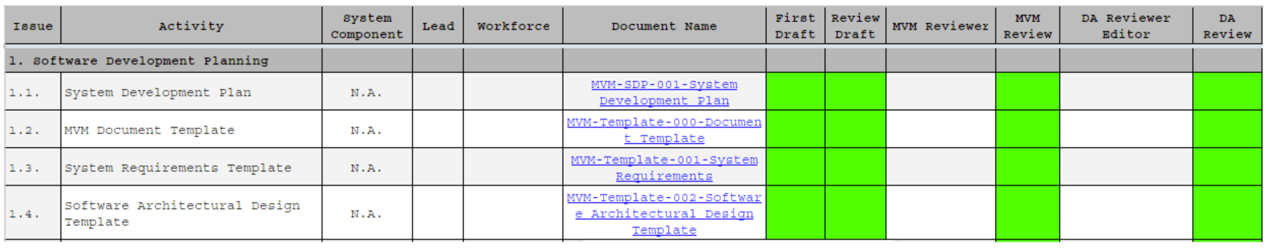
3. *Software requirements analysis* and *Software detailed design*, which are performed for each of the components previously identified during the architectural design phase;

4. *Software implementation* and *Unit testing*, which are performed for every component of the MVM;

5. *Integration and Validation testing*, which are performed by integrating all software components with the hardware and by testing the system as a whole.

For each phase of the process, the software development plan defines the tasks to be performed, the inputs necessary for its execution, and the deliverables expected when the phase is completed. As shown in Fig. 13, the adopted process model recalls a V-model, in which all the activities required by the standard are mapped to process phases. For each software component, the identified development process allowed the teams to work iteratively (inner dark circle in Fig. 13) to guarantee the conformance among requirements, architecture, design, and implementation. The integration between the V-model and agile practices favored flexible responses to changes, due to the need to develop the ventilator as quickly as possible, caused by the fact that the work on the MVM was performed during an emergency. In this way, the MVM team could better parallelize the process in an agile-like mode and foster a collaborative approach. Moreover, agile practices are considered by the outer circle in Fig. 13 as well: after validation/system testing all the processes can be re-executed one or more times to integrate solutions to detected problems. At the end of the day, the MVM team combined various models of software development processes, namely the V-model with agile practices and model-driven development (mostly for the state machine component, such as the one reported in Fig. 8).

Lesson learned 1: 1. *IEC 62304 and V-Model:* The development process was strongly influenced by the IEC 62304 standard, so the V-model, although not mandated, is the "best fit" with regulatory requirements as it produces the necessary deliverables required when seeking regulatory approval. 2. *Use of agile practices:* However, it was necessary to integrate the V-model with agile practices, to combine efficiency, quality, maintainability, and flexibility.



Issue	Activity	System Component	Lead	Workforce	Document Name	First Draft	Review Draft	MVM Reviewer	MVM Review	DA Reviewer Editor	DA Review
1. Software Development Planning											
1.1.	System Development Plan	N.A.			MVM-SDP-001-System Development Plan	🟩	🟩		🟩		🟩
1.2.	MVM Document Template	N.A.			MVM-Template-000-Document Template	🟩	🟩		🟩		🟩
1.3.	System Requirements Template	N.A.			MVM-Template-001-System Requirements	🟩	🟩		🟩		🟩
1.4.	Software Architectural Design Template	N.A.			MVM-Template-002-Software Architectural Design Template	🟩	🟩		🟩		🟩

Figure 14: Excerpt of the task list.

Teams definition and meetings planning

During this activity, the teams have been defined, as well as their composition. Seven subgroups were created ad-hoc during the project, taking into account competencies, workload, and availability of individuals. For each group, the coordination team identified a group leader, which was responsible for meeting deadlines and for reporting the progress to the project coordinator, and an activity to be assigned to each of them. The first group was in charge of defining the software development plan, supporting activities, and performing the risk analysis; the second group had the goal to define software requirements; the software architecture specification was designed by group three, while group four developed the MVM software. Groups five and six run software unit testing, integration and system testing, respectively. Finally, another group prepared the operational and maintenance manual.

Then, the coordinators have defined the structure and schedule of the meetings. The entire MVM team met daily (including holidays) for 1 hour from 5 PM to 6 PM CET to check the status of the project and set the goals for the next day. The project leaders went through the task list (see Fig. 14) to check the status of each task and to check if some obstacles emerged. Moreover, each sub-team had to organize task-specific meetings to synchronize their work at other times of the day. Because of the lockdown due to the COVID-19 emergency in Italy, only a few in-person meetings were organized (and authorized by the special national commissioner). Moreover, since the teams included people (mainly) from Europe and America, only the late afternoons and nights were suitable for online meetings.

The aspect that made the MVM project particularly challenging was related to the characteristics of the team. In fact, the teams were multidisciplinary and heterogeneous, involving people with different backgrounds, including physicians, physicists, electronic engineers, and computer scientists, and composed of volunteer people motivated by the social nature of the project and by their passion.

Lesson learned 2: 1. *Coordination effort*: The project was successful, but there was a quite huge overhead of coordination, with various calls every day from the morning to evening. The coordination of the team should not be underestimated. Open-source software development could be a good development experience from which projects of this nature can learn. 2. *Enlarging team*: The experience on the development of the MVM has confirmed what is reported in the Mythical Man-Month [49]: adding people is not necessarily a good solution to improve the efficiency and effectiveness of a team. 3. *Commitment and participation*: Having responsibilities for each sub-activity and setting strict intermediate goals have favored commitment and participation.

Supporting activities

In parallel to the definition of the development process, a set of supporting activities has been identified: project management, the definition of a change control process, the definition of the development environment, and the definition of code guidelines. These activities are related to the production of documents, guidelines, and processes to be used throughout the software life cycle.

Initially, the teams selected the *tools* supporting them in project management. The management team initially had evaluated the use of a project management tool like Jira, but in the end, decided to use only a combination of more accessible tools, considering the heterogeneity of the group and the non-expertise of all the members with dedicated software, like Google Drive, GitHub, Zoom, and Slack. For modeling system requirements and system architecture, UML (Unified Modeling Language) has been used, and it turned out to be understandable also for people not experts in software engineering. We have put all the documents on Google Drive and the code on GitHub, which provides hosting for software development and version control using Git. This allowed people from all over the world to contribute to the development of different parts of the software. This approach has allowed keeping track of all the changes in the code, which is really useful to manage and control the software development process. Moreover, GitHub has also been used to manage issues and to signal addings or fixes to the software. Furthermore, a continuous integration system has been established using Travis CI[1] (Continuous Integration).

For all the activities that foresee a document as output, the team decided to apply two *review* steps. The first was performed internally by a designated member of the team. Once the document was approved, the Design Authority (Elemaster - one of the companies involved in the MVM production) was in charge of reviewing such documents and producing the Review Acceptance document containing all the comments. Later, the comments were addressed and possibly included in a new version that was

[1]https://travis-ci.org/

resubmitted again to the Design Authority. This process was performed iteratively until the Design Authority approved the final version of the document.
Considering that several activities of the software development life cycle were supposed to produce a document as a result, the coordination team needed to produce *document templates*. The initial templates for the documents were kindly provided by the Canadian Nuclear Laboratories (CNL) sub-team, which has great experience in critical software certification (not in the medical field, though). This greatly helped to speed up the process from the start. To keep track of the links among requirements in the documents, during the development of the MVM, a CNL collaborator developed an in-house tool. This tool reads each document subject to traceability and generates a traceability matrix for the requirements. The tool has been daily executed to illustrate the existing links and report missing or faulty links.

Lesson learned 3: 1. *Multiplicity of tools:* The use of a great variety of tools (one tool for each particular purpose) even if not integrated and not specific for software project management, has provided indispensable support to the team. 2. *Templates and review process:* Having a partner that provided all the necessary templates and a clear review process has helped to define which activities should be performed. 3. *Use of UML:* Standard graphical notations like UML have shown to improve communication and to be easily usable by non-software experts (very skilled in other fields, though) too.

3.3.2 Development phases

After having performed the activities described above, which generally refer to the development process, we have tackled the activities required by the process presented in Fig. 13. In the following, activities and lessons learned on each phase of the development process are presented.

System requirements (5.2)

One of the obstacles encountered during the first implementation of the MVM was the lack of well-defined and traceable system requirements. This caused a lot of confusion between the developers of different components, and some operations did not meet the requirements of the standards that regulate the ventilator development.
According to point 5.2 of the IEC 62304 [64] standard, the system requirements analysis has been performed to define the requirements of the device at a higher level of abstraction. In this phase, the team has defined functional, performance, safety, and cybersecurity requirements, the overall

structure of the system, environmental conditions, materials, and human factors. Furthermore, each requirement has been uniquely identified by a number, which has been used for traceability purposes during the whole software development process.

Writing complete software requirements has allowed the MVM team to identify conceptual bugs in the first prototype of the ventilator. For example, the MVM team has discovered that the prototype was faulty because, in case of failure of one of the components, the system was not able to put itself in a safe mode, so that the valves are positioned to allow the patient autonomous breathing. For this reason, the necessity to have an additional component, namely the *supervisor*, came out. This required a change of the initial electronic board with the introduction of a small microcontroller devoted to assuring the safety of the device.

During the writing of system requirements, reverse engineering some parts of the prototype has been useful in order to get a complete and consistent requirement specification of the device operation. It has been applied when no enough information was available, and this was the case, for example, for the specification of the alarms. The reverse engineering process also helped to reveal details useful to specify the duration of a trigger window, namely the time interval within which spontaneous breathing can be detected (in PCV mode - see Chapter 2 for more details on the ventilation modes).

Lesson learned 4: 1. *Written requirements:* Not having written requirements since the beginning led to various attempts to address the requirements in different software components. For this reason, precise system requirements are also very important in an emergency situation. Having developers referring to the same written documents without inconsistencies reduces the development time. 2. *Reverse engineering:* For systems for which a prototype is available, especially if it is developed by domain experts, reverse engineering has shown to be a viable solution to discover the functionalities and configuration parameters to be included in the requirements of the system. 3. *Need of a traceability system:* A traceability system helps developers to trace all the requirements and their changes throughout the development process.

Software Architecture Design (5.3)

Based on the requirements identified during the *System requirements analysis* (see Sect. 3.3.2), the architecture of the MVM has been designed to be composed of three main software and hardware units, namely Graphical User Interface (GUI), Control Software (Controller), and Supervisor Software (Supervisor). The overall representation of the Software Architecture is reported in Fig. 6.

The GUI is the software running on the touch screen panel. It displays information to the doctors like airway pressure, minute volume, positive-end expiratory pressure (PEEP) for the most recent breath, respiratory rate (RR), and peak inspiratory pressure. Furthermore, the user can use it to set ventilation and alarm parameters and thresholds. On the other hand, the controller receives user input from the GUI, e.g., the start/stop ventilation command. It implements the state machine of the ventilator behavior (see Fig. 8), which has mainly three regions: ventilation off, ventilating in PCV mode, and ventilating in PSV mode. Based on the current state, it opens/closes the input and output valves. In addition, the controller is responsible for managing ventilator alarms in case of errors.
Finally, the supervisor monitors the overall behavior of the system, checks if the controller, the GUI, and the hardware are working as expected, and, in case of errors, it raises alarms. Furthermore, the supervisor forces the machine into a safe mode to prevent patient injuries in the event of errors of the other units during ventilation.

Lesson learned 5: 1. *Upfront aspects balancing:* As we can learn from software architecting, it is important to go towards "just-in-time architecture" [145] and to find a balance between upfront aspects (what is planned before the start of development) and emerging aspects (what appears as decisions are taken in the course of the development, e.g. by fixing wrong assumptions or making decisions deliberately postponed) [179]. 2. *Importance of the architecture:* Software architecture is still important even during emergency development. In fact, the development team has experimented that without a well-defined architecture (as for the prototype), it was not clear how software components were supposed to synchronize and exchange information among them.

Risk management (7)

The IEC 62304 standard requires each component to be classified based on the potential to cause injuries. Table 6 reports the safety class for each of the three components of the MVM. In particular, both the GUI and the controller are classified as class A software because their behavior, despite affecting the operation of the machine, does not cause patient injuries, since safety-critical tasks and

Software unit	Safety class
GUI	A
Controller	A
Supervisor	C

Table 6: MVM software units and their safety classifications.

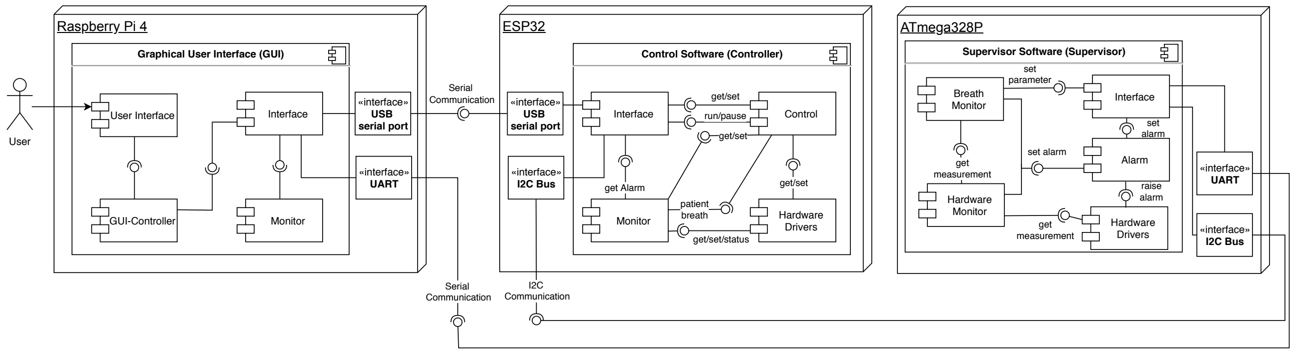


Figure 15: Detailed MVM software architecture.

decisions are previously approved by the supervisor. On the other hand, the supervisor is the most critical component, since it both forces the machine into a safe mode to prevent patient injuries in case of errors during ventilation and intervenes in case of GUI and controller failures, so it is classified as class C.

Lesson learned 6: 1. *Safety assurance effort:* Isolating safety-critical features, by organizing the system in different components, has allowed the testing team to focus the safety assurance effort on a limited portion of the system.

Software Requirement Analysis (5.2) & Software detailed design (5.4)

After having identified the components and their safety class, the third activity in the development process consists of specifying for each software component the requirements in a separate document, detailing those introduced in the system requirements. The documents of *Software Requirement Analysis* and *Software detailed design* contain different types of requirements, among which there are functional and capability requirements, software inputs and outputs, interfaces between software and other components, alarms and warnings, user interface requirements, and requirements related to system installation and maintenance. In order to increase the understandability, also for team members not used in reading software requirements, state machines have been widely used to define the behavior of GUI, Controller, and Supervisor. Alternatively, the teams have used diagrams and drawings, and in few cases a more powerful tool, such as Yakindu (more in Sect. 3.3.2), which has allowed to identify both states and events that trigger the change of state. For each state in every diagram, the MVM team has defined the detailed behavior, the user inputs, the expected outputs, performance, and failure conditions. Furthermore, for each software component, the software unit interfaces have been defined, to ensure that the software subsystem will communicate properly with external components. In Fig. 15 a more detailed architecture of the software components of MVM is reported.

The doctors interact with the User Interface sub-component, and the interaction is handled by the GUI controller and the Interface component, which manages the connection with the control software - via a USB serial port - and with the Supervisor - via a UART interface. The monitor module is used to supervise the interaction with the GUI.
The Control sub-component implements the logic of the Control Software (Controller). It is in turn composed of two sub-components: the valve controller and the state machine. The former controls the valves, while the latter controls the transition between operation modes (i.e., ventilation off, running in PCV mode, and running in PSV mode). The controller receives user inputs from the GUI, e.g. the start/stop ventilation command. The vital signs of the patient are checked by the monitor sub-component, and, in case of errors, ventilator alarms are raised by the controller. The controller also interacts with the hardware through the Hardware Drivers component to open or close the input and output valves.
Finally, the Supervisor Software (Supervisor) component gets measurements from the Hardware Drivers component, monitors both the hardware devices and the patient breath, and, when needed (e.g., when switching to the safety mode), raises alarms and changes parameters of the controller and the GUI.

Lesson learned 7: 1. *Modularity and parallelization:* Designing a product in a modular way has been a successful decision, since, in a distributed project (such as the one of the MVM) it has allowed different teams to work in parallel on different parts of the system. 2. *State machines for wide interpretability:* Using state machines, for specification and design, has contributed to favoring the discussion on the adopted solutions, even with people not used to software development, since graphical representations are easily understandable.

Implementation (5.5)

Due to the necessity of adhering to the certification standard, several parts of the software which were already available before the re-engineering effort needed to be changed. Table 7 reports the lines of code of the three main software units (before the re-engineering effort and at the end).
The *GUI* unit is written using the PyQT5 framework in Python. It is a set of Python bindings for Qt5, which allows access to many aspects of modern desktop and mobile systems[2]. In addition to the functionalities already present in the prototype, several new functionalities expected by the software requirement specifications have been added. For example, many alarms have been changed, in terms

[2]More info at: `https://pypi.org/project/PyQt5/`

	Lines of code		
Software unit	**Prototype**	**Released version**	**Language**
GUI	14,347	26,027	Python
Controller	4,653	14,331	C++
Supervisor	NA	2,689	C++

Table 7: Lines of code of the MVM software units.

of thresholds and behavior. Moreover, to avoid problems in the communication between GUI and controller, a new protocol used by the GUI to send/receive messages to/from the controller, and implementing the guidelines defined in the IEC 61784 standard [66], has been devised.

The majority of the re-engineering activities have been performed over the *controller* unit:

- The *state machine* sub-component has been completely rewritten. The development team has introduced the use of the Yakindu Statechart tool[3] for its implementation, as shown in Fig. 8. In detail, after the startup and the self-test phases, the machine is put in the ventilation-off mode. From there, it can go either to the PSV or PCV modes, depending on the choice of the doctor and on the patient's condition. Inside these modes, there are other sub-states (including inspiration and expiration). After having modeled the state machine component, its C++ code has been automatically generated from the Yakindu model.
- The *valve controller* sub-component has been modified, accordingly to the hardware modifications. For example, a new controller tuning method has been implemented.
- The *alarms* have been updated and adapted since they have to comply with those that have been added or modified in the GUI code. In particular, all the alarms have been implemented to comply with the IEC 60601-1-8 [63] and ISO 80601-2-12 [106] standards.

As previously introduced, in the re-engineered MVM version, the *supervisor* has been developed in C++ from scratch, and it required the addition of two different software serial lines: one used to communicate with the GUI and one with the controller.

Lesson learned 8: 1. *Mix of programming languages:* Using several programming languages in a single project is usually discouraged [128]. However, in an emergency (such as during COVID-19) in which the products have to be delivered as soon as possible, we have experienced that having more languages allows the inclusion of more developers and speeds up the implementation process, with only a minimal effort in the integration of the code. 2. *Coding standards and guidelines:* Sharing the coding standards and guidelines (e.g., the importance of

[3]`https://www.itemis.com/en/yakindu/state-machine/`

comments [150]) with all the people involved in the implementation phase is of key importance, in particular with heterogeneous development groups, even during emergency development.
3. *Advantages of state machines in implementation:* State machines added flexibility and maintainability since it was very simple to modify them and then regenerate code, which was directly integrated, through a wrapper, into the hand-written code.

Unit Testing (5.5)

Aiming at introducing CI/CD techniques in the development process, a continuous integration system on Travis CI has been configured to guarantee that new software implementations did not compromise the functioning of the already existing code. This way, every commit made on each component brought forth a new re-execution of all the unit tests. In fact, the testing activities have been executed in parallel with the implementation, since every test failure has required corrections in the code, and it has been performed against the unit software requirement specifications.
As MVM has different components, each written in a different programming language, its testing has been performed using several testing frameworks. The testing team has unit-tested the GUI using PyTestQt[4], which has allowed to simulate users by faking clicks on the buttons and testing that the behavior of the GUI was the one expected. To test the controller and the supervisor, the hardware has been mocked to emulate the interaction with it. Controller and supervisor test cases was written using the Catch2 framework[5] and the Trompeloeil mocking library[6].
All the testing failures have been tracked using the GitHub issues tracking system, and this has allowed the testing team to monitor the progress of the fixes in the software.

Lesson learned 9: 1. *Testing not only safety-critical components:* Defining in advance the safety classes of all components of the developed system can significantly increase the speed of testing activities. In fact, medical software safety standards do not mandate extensive unit testing for class *A* components. Thus, a good practice is to design the system in a modular way, isolating all the non-dangerous functions (i.e., in class *A*) that testers can limitedly check.
2. *Importance of testing:* Besides what is required by standards, testing activities are important when performed for safety-critical components. This is a consolidated aspect when working with software engineers but not for all the people composing heterogeneous teams such as the

[4]`https://pypi.org/project/pytest-qt/`
[5]`https://github.com/catchorg/Catch2`
[6]`https://github.com/rollbear/trompeloeil`

MVM ones. 3. *Advantages of CI tools in community projects:* As MVM has been a community project, where a lot of people have worked at the same time on the same system, CI tools have proved to be crucial for maintaining under control the modifications made by all developers.

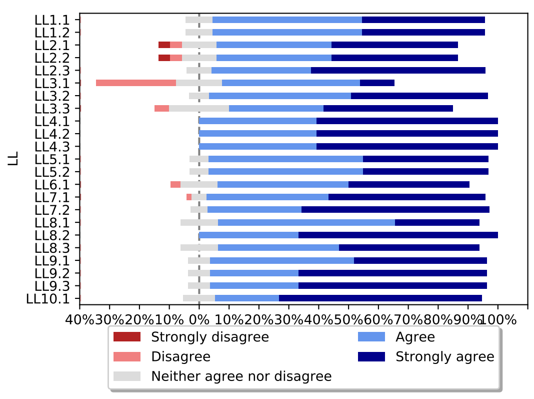
Integration (5.6) & Validation Testing (5.7)

During integration and validation testing, hardware and software components have been incrementally integrated. While developing the MVM, the main challenge has been the need of having the hardware available, since some of the software components included in the controller or supervisor require direct interaction with it. This is particularly true for the final integration steps, while for the first integration phases the HW can be ignored and just simulated. Thus, final integration testing phases have been performed on-site, only by the people working in the company that produces the MVM, or by the ones that had a physical version of the ventilator.

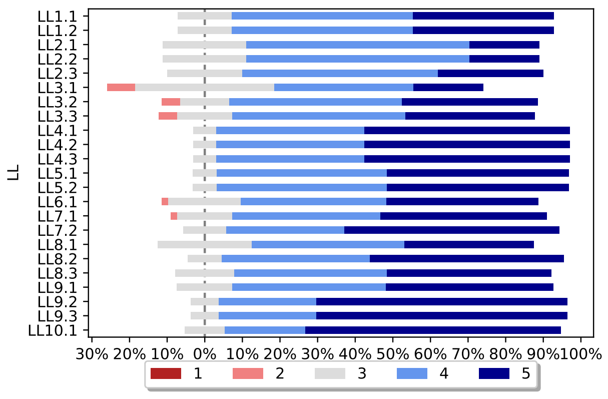
During the development of a medical device that interacts with patients, one should simulate not only hardware and interaction with the doctors but also the patients. For this purpose, an active lung simulator (such as the ASL 5000 [70]), or a passive mechanical one, has been used. The results of the integration testing activities, for each one of the test cases, have been reported in Integration Test Procedure and Integration Test Report documents.

With validation testing, the whole system has been tested as a unit, to confirm the correct behavior of MVM according to the system requirements. This activity must be performed over a real physical version of MVM, without any hardware simulation, and simulating patient breath using the ASL 5000 active lung simulator. It has been guided by the ISO 80601-2-12 standard [106], to prove the basic performance and usability of the ventilator.

Lesson learned 10: 1. *Integration testing for SIMDs:* It is particularly challenging to develop and validate software-in-medical devices (SIMDs) and, in general, systems that integrate hardware, software, and mechanics by distributed teams. Often, real hardware is needed for testing the software that is affected or affecting a piece of hardware. Software-in-the-loop simulation is often a good solution to this challenge; however, it is not a solution, in general. In fact, simulation requires a special setting with professional simulation tools and an accurate hardware model. They are not always readily available, especially in a context in which the hardware is under development as the software is. Furthermore, as we can learn from robotics, "*Cur-*



(a) **Agreement with each lesson learned.**



(b) **How much is the LL important?**

Figure 16: Importance of the lessons learned.

rent simulation solutions are not capable of emulating real-world phenomena in a sufficiently realistic manner. " [86].

3.3.3 Validation of the lessons learned

After having defined the lessons learned (in the following, referred to as LL), their validation has been performed with a variety of experts (appertaining to the Italian engineering society) during two seminars, by following these three steps:

1. Additional questions have been asked to investigate the opinions of experts before presenting lessons learned;
2. The lessons learned have been presented;
3. Each expert evaluated the importance and agreement with the presented lesson learned.

The percentages of agreement/disagreement with the presented LL are reported in Fig. 16a. The results gathered show that, generally, the participants (strongly) agreed with the LL, except in a few cases. In particular, some participants disagreed with LL.2.1 (Coordination effort) and LL.2.2 (Enlarging team). The former because they were skeptical of using open-source software instead of more specific ones, and the latter because they thought that having as many people as possible would lead to faster results. The opinions received highlighted that it was confusing to ambiguously talk about teams without clearly distinguishing between coordination and development teams. LL.3.1 (Multiplicity of tools) created the most disagreement, mainly because it was not clear the context where different tools would be used. Many participants did not agree with the use of UML (LL.3.3) because other visual notations could be used, even if they are not recognized as a standard in the community. For

LL.6.1 (Safety assurance effort) and LL.7.1 (Modularity and parallelization) there were only few disagreements. Finally, for LL8.1, the validation highlights that the use of multiple programming languages can lead to integration problems.

The behavior of Fig. 16a is reflected in Fig. 16b. Not surprisingly, for some participants, lessons learned with disagreement in Fig. 16a are also not that important: LL.3.1 (Multiplicity of tools), LL.3.3 (Use of UML), LL.6.1 (Safety assurance effort), and LL.7.1 (Modularity and parallelization). Exceptions are LL.2.1 (Coordination effort) and LL.2.2 (Enlarging team) for which many participants did not agree, although they retain them important issues. On the opposite, they agree/strongly agree with LL.3.2 (Templates and review process), but some of the participants do not think that it is important for critical software development under emergency.

3.4 Guidelines

Exploiting the feedback received during the validation of lessons learned, in this section, I present the guidelines for developing medical devices under emergency derived from each lesson learned. For deriving guidelines, lessons learned have been grouped, filtered, and modified. Table 8 reports how the validation activities have been used to obtain the corresponding guideline.

Lessons learned	**Validation result**	**Guidelines**
LL.1.1 *IEC 62304 and V-Model* **LL.1.2** *Use of agile practices*	Merge the two lessons learned LL.1.1 and LL.1.2 into a unique guideline.	**GL1** *Plan-driven/predictive and agile integration*
LL.2.1 *Coordination effort* **LL.2.2** *Enlarging team* **LL.2.3** *Commitment and participation*	Distinction between coordination team and development team in terms of responsibilities and activities.	**GL4** *Resources initial estimation* **GL5** *Coordination team and plan* **GL6** *Responsibilities assignment* **GL7** *Flexible development teams*
LL.3.1 *Multiplicity of tools*	Distinction between tools for coordination and communication of the coordination team and of development teams.	**GL8** *Inter-team coordination and communication* **GL9** *Intra-team coordination and communication*
LL.3.2 *Templates and review process*	Clarification of what to do when existing templates are not available and a review process is not already established.	**GL2** *Review process* **GL3** *Documentation templates*
LL.3.3 *Use of UML*	Generalization to visual/graphical notation in general	**GL11** *Use visual and graphical notations*

LL.4.1 *Written requirements* **LL.4.2** *Reverse engineering*	Merge the two LLs.	**GL12** *Precise requirements and reverse engineering*
LL.4.3 *Need of a traceability system*	Recommendation of a practice (traceability system) that was not extensively used during the initial development of the MVM but deemed to be important.	**GL10** *Define a traceability system*
LL.5.1 *Upfront aspects balancing* **LL.5.2** *Importance of the architecture* **LL.7.1** *Modularity and parallelization*	Clarification of how to manage changes in software architecture and introduction of communities of practice as an instrument to evaluate the impact on architecture and to assess and validate architectural decisions.	**GL13** *Define an architecture upfront* **GL14** *Limit the upfront architecture to stable decisions* **GL15** *Update the architecture* **GL16** *Exploit communities of practices*
LL.6.1 *Safety assurance effort*	Reformulation of the lesson learned.	**GL17** *Isolate safety-critical parts*
LL.7.2 *State machines for wide interpretability* **LL.8.3** *Advantages of state machines in implementation*	Identification of the two main uses of state machines.	**GL18** *Use state machines in specifications* **GL19** *Use state machines for code generation*
LL.8.1 *Mix of programming languages*	Specification of some caveats (e.g., when it does not make integration difficult).	**GL20** *Different programming languages*
LL.8.2 *Coding standards and guidelines*	Reformulation of the lesson learned.	**GL21** *Coding standards and guidelines*
LL.9.1 *Testing not only safety-critical components* **LL.9.2** *Importance of testing* **LL.9.3** *Advantages of CI tools in community projects* **LL.10.1** *Integration testing for SIMDs*	The lessons learned overlapped in some way and the research team has identified the need for a better organization of them. This led to the reformulation of the lessons learned in three more clear guidelines.	**GL22** *Continuous integration and unit testing* **GL23** *Focus on testing activities* **GL24** *Role of emulators*

Table 8: Mapping between lessons learned and guidelines for developing medical software under emergency.

Overall, based on the validation of the lessons learned, in the guidelines, a clear distinction has been made between the coordination team and the development team. The former is more structured and

stable, while the latter is more agile and dynamic. Furthermore, the validation of the lessons learned has been used as precise and specific feedback in the formulation of the guidelines.
As well as for the lessons learned, the guidelines are divided into those referred to specific phases of the development process and those concerning the process in general.

3.4.1 Development process

GL1 Plan-driven/predictive and agile integration: *Integrate plan-driven/predictive processes with agile practices, to combine rigorousness with efficiency and flexibility.* The integration of different processes allows one to benefit from the good characteristics of both plan-driven [21] (or predictive [133]) and agile processes. A mix of different development processes has already been proposed in the literature. For example, the pros and cons of blending agile and waterfall processes are discussed in [115, 149], while in [132] the authors integrate agile and the V-model.

GL2 Review process: *Define a clear review process to identify the activities that should be performed.* A clear review process allows to speed up the development since when a review is required before continuing with the next activities, the steps to follow are already well known and it does not become a bottleneck. It also offers an efficient way to improve the quality and effectiveness of the development process [110].

GL3 Documentation templates: *Reuse and/or adapt existing templates, when available, for producing the documentation required by certification standards and processes, otherwise produce precise templates to be adopted by the entire project.* Considering that saving time is important, especially under emergency, it is better if there are already established templates that can be reused (for example, given by a collaborating company). This aspect is usually underestimated, especially for agile-based software development processes [122] that claim not to produce a lot of structured documentation. However, for safety-critical software, the regulations require producing a great amount of deliverables that cannot be avoided. If templates are not available, it is recommended to define them before starting the development process in order to know from the beginning which information must be reported.

GL4 Resources initial estimation: *Estimate the competencies, resources, and commitment of the various team members in the initial phases of the project, since potential new members should be added in this initial phase.* Despite the emergency, in fact, it is important to invest some time in this upfront activity during the startup phase, as introducing members during the development process could increase the time-to-market due to the time required to understand the project. This guideline is confirmed by many other researchers, such as in [49] and [127].

GL5 Coordination team and plan: *Define the coordination team and the coordination plan of the team upfront and in the startup phase.* Despite the emergency, the coordination team must be ready to work as soon as possible and there must be clear indications for the development teams. The lack of precise indications could lead to misunderstandings and increase the time required to complete the project. The same conclusions have been observed by [135] and [164], in which the importance of coordination teams is highlighted.

GL6 Responsibilities assignment: *Assign precise and stable responsibilities to the members of the coordination team.* The coordination team should be as stable as possible to avoid delays that can easily propagate to the development teams. The importance of building a coordination team is well known in the literature. In particular, most of the research works consider the presence of effective leaders who both steer development and motivate developers crucial to ensure a successful product [121].

GL7 Flexible development teams: *Development teams can be created according to the needs during development and members should be prepared to help in various tasks according to their availability and competencies. In every iteration or sprint, team members can be assigned to different tasks.* Development teams should work agile and change should be the norm rather than the exception. Under emergency, there are strong timing constraints and agility and flexibility within the development team can greatly help. The idea of flexible development teams is widely adopted in agile processes, especially when the software has to be produced for an emergency, e.g. the COVID-19 pandemic [170].

GL8 Inter-team coordination and communication: *Define the communication and coordination instruments, tools, and protocols for inter-team coordination.* When working in different groups, there is the need for clear and stable instruments, tools, and protocols for communication and coordination among the different teams. Many of the responses received from the surveys about LL3.2 (see Sect. 3.3), i.e., the lesson learned from which this guideline derives, complained about the use of multiple tools, instruments, and protocols. For this reason, the guidelines isolate the inter-team coordination, for which the coordination mechanisms must be fixed, and the intra-team communication (see *GL9* for further details). Each project may require specific mechanisms for inter-team coordination, such as the one proposed in [142].

GL9 Intra-team coordination and communication: *Delegate to the members of the development team the selection of instruments and tools for development and intra-team coordination and communication.* Within the development teams, members should be able to select the development, communication, and coordination instruments and tools they like. This could be considered counter-intuitive, but it can help in saving a lot of time, since no training in a specific instrument or tool is necessary for the development team members. Note that coordination and communication inter-team must be fixed a priori, as defined by *GL8*. Guaranteeing a certain grade of autonomy is a well-established principle in agile-based development teams, even if there are still a lot of challenges to be faced when implementing autonomous teams [163], mainly related to inter-team tasks.

3.4.2 Development phases

GL10 Define a traceability system: *Define upfront and in the startup phase a traceability system for the entire development.* Traceability is of key importance in the development of safety-critical systems, both for security and certification purposes. Then, from the initial phases of the project, there is a need for a traceability system for the entire development. In fact, even under emergency, traceability information can be used to support the analysis of implications and integration of changes that occur in the system, its maintenance and evolution, and its testing activities. This guideline is recognized as valid from several works in the literature, such as [162] and [99].

GL11 Use visual and graphical notations: *Use, when possible, visual, graphical, and easy-to-understand notations (e.g., UML) for communication among team members with heterogeneous expertise and competencies.* The importance and wide interpretability of graphical notations is universally recognized. This is even more important during an emergency, when teams can be composed of members with different backgrounds and knowledge. In this case, graphical notations might facilitate communication among team members [59].

GL12 Precise requirements and reverse engineering: *Write precise system requirements also in an emergency situation. If a prototype exists, use reverse engineering to extract useful information.* Requirements are important to guide the development and the validation phase. Even if in agile processes is common to skip or not to focus on requirement specifications, although under emergency, the requirements must be written in a precise way. This guideline is particularly useful for safety-critical systems since certification authorities require a set of documents among which there are the requirements. The necessity of precise (even formal) requirements has been advocated for a long time, for example, in [8, 143].

GL13 Define an architecture upfront: *Define an architecture upfront to allow different teams to work in parallel on different parts of the system and facilitate integration.* When working under emergency, every aspect that could increase the rapidness of the development is important. For this reason, a clear identification of components and interfaces might help development teams in being faster, by working independently and in parallel. The upfront architecture should be as stable as possible (see *GL14*), but teams must be prepared to adapt it in the event of a change in requirements [76].

GL14 Limit the upfront architecture to stable decisions: *Limit the upfront architecture to stable decisions while paying attention to concerns that matter across team borders.* An architecture description can be considered a boundary object between multiple cross-functional teams: it can be used to create a common understanding across sites while preserving each team's identity [179]. For this reason, an upfront architecture should be limited to stable decisions, and then it should be updated with emerging aspects. The upfront architecture should include aspects that influence coordination and those interfaces that are shared among different teams (see *GL13*).

GL15 Update the architecture: *Integrate system architects into teams to capture emerging aspects during development and update the architecture accordingly.* Since the upfront architecture should be limited to stable aspects, architects, or those team members playing the role of architects and taking architectural decisions, should capture emerging aspects during development and update the architecture accordingly [180].

GL16 Exploit communities of practices: *Exploit communities of practices to reason about changes that impact the architecture, and to assess and validate architectural decisions.* In order to reduce assumptions that can become inconsistencies, it is important to carefully assess decisions before setting them into stone. Community of practices is a good instrument to enable architects to reason about changes [179]. They can be effectively used to solve issues that span over multiple teams [108].

GL17 Isolate safety-critical parts: *Isolate safety-critical features in specific components or modules to focus the safety assurance effort on a limited portion of the system.* Safety standards often require different levels of attention and different validation activities. The isolation of safety-critical features in specific components or modules permits limiting the validation and certification activities, and therefore saving time under emergency development. The aim is similar to the well-known practice of using a security kernel with the desire to isolate and localize all "security-critical" software in one place [156].

GL18 Use state machines in specifications: *Use state machines to specify modes and mode transitions in the requirement specification.* State machines are used in various domains and are good tools for easily communicating complex behaviors, especially when development teams are heterogeneous. Moreover, if some kind of formal verification is needed, with state machines (or equivalent methods), it can be easily performed [7].

GL19 Use state machines for code generation: *Use, when possible, executable state machines to specify the main functional logic and the critical part of the system, and then generate code from them.* Based on the MVM experience, the use of executable state machines promotes modifiability, maintainability, and understandability (see *GL18*). In particular, tools like Yakindu SCT[7] or other state-machine-based tools can be a good choice when it comes to developing a critical part of the system, as they allow generating automatically actual code [25, 31], which can be verified and tested at state-machine-side [172].

GL20 Different programming languages: *Allow the use of different programming languages to facilitate the inclusion of heterogeneous developers and speed up the implementation process, when this does not create integration problems.* When the use of different programming languages does not create integration problems, e.g. when code produced with different languages is deployed on different hardware components or when the communication mechanisms between modules are language-independent, it would be beneficial to allow development teams to use languages they are familiar with [128]. As explained in LL8.1 in Sect. 3.3, when developing software under emergency, with heterogeneous development teams, exploiting the competencies that every member has in a particular programming language can aid in reducing the time required for the completion of the project.

GL21 Coding standards and guidelines: *Adopt coding standards and guidelines from the beginning.* Safety standards often require coding standards and, in general, they promote the quality of the code. Using coding standards and guidelines can increase the readability of the code, which is important for code inspections and static analysis. Moreover, even under emergency, if the composition of the development teams is heterogeneous, defining in advance coding standards and guidelines is useful for preventing possible errors or hardly-understandable code. The importance of adhering to coding standards from the beginning is highlighted by many researchers, such as in [40], where the authors empirically assess their value and suggest not to insert them at a later time as any modification (aimed at adapting the software to the chosen coding standards) has a non-zero probability of introducing a fault or triggering a previously concealed one.

GL22 Continuous integration and unit testing: *Use CI tools and automated unit testing in order to continuously integrate the contributions of the various teams, to keep and promote quality, and to maintain under control the modifications made by all the developers.* CI tools and automatic testing instruments enable various teams to work in parallel without breaking the code and permit to avoid the big bang integration problem. The importance of continuous integration is highlighted in [74], where CI tools are claimed to be effective, since they allow a shorter time between the possible introduction of a bug in the system and its detection. This is of paramount importance, especially for complex safety-critical systems developed under emergency and in a distributed way.

[7] https://www.itemis.com/en/yakindu/state-machine/

GL23 Focus on testing activities: *While guaranteeing the quality of the entire system, testing activities should focus on safety critical components as required by the standard.* Safety critical components are those that require major attention, in terms of quality and test effort. Parts of the system that are not critical can follow classic quality management recommendations. Focusing software testing activities on critical components is often used in practice, especially when companies want to reduce their time-to-market [79]. Anyhow, testing is important on all components of a safety-critical device, regardless of the safety classification.

GL24 Role of emulators: *When possible, use simulators and/or emulators, but plan for integration, system, and acceptance testing phase.* Simulators and emulators can speed up development and validation, but are often limited and integration, system, and acceptance testing cannot be performed using them. This is a well-known aspect of integration testing for safety-critical systems, such as for trains and ships [72], or automotive [159]. Especially if the system under development is composed of hardware and software, the *testing in the field* activity must be planned and performed in the real environment, with the real hardware.

3.4.3 Validation of the guidelines

Validation via questionnaires

After having defined the guidelines, they have been validated using two different ways, namely through questionnaires and interviews as explained in Sect. 3.4.3.

During the questionnaires, the participants have been asked about their agreement with each guideline, using a Likert scale (from strongly disagree, to strongly agree). The results of these surveys are presented in Fig. 17. The agreement level for each guideline is generally lower than the one on the lessons learned (see Fig. 16a). This was somehow expected since the guidelines are given in a more affirmative and prescriptive style. Moreover, this is also due to the wider experience and expertise of the involved experts during questionnaires which have been involved in order to collect different points of view. However, in all cases (except for GL20), the agreement levels remain very positive.

Validation via interviews

To further investigate the results obtained during the questionnaires, some of the participants have accepted to be interviewed (see Sect. 3.4.3) to examine the reasons for which they were particularly agreeing or disagreeing with the proposed guidelines.

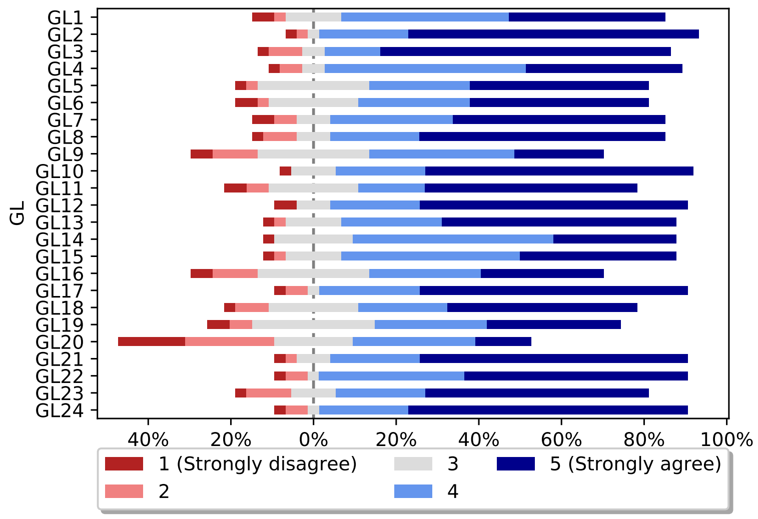


Figure 17: How do you agree/disagree with the GL?

Tab. 9 and Tab. 10 summarize the opinions collected in terms of the benefits expected if one follows the guideline and the risks. Especially for the guidelines with the lower agreement (like GL20), thanks to the direct interaction with the experts, the summary tables better identify the limits and circumscribe their intended use. This will enable users of the guidelines to carefully assess how each guideline should be applied to their project. For instance, for GL20, I here report a typical scenario in which it can be followed, i.e., when different programming languages are used on different hardware components.

Guideline	Benefits	Risks
GL1 *Plan-driven/predictive and agile integration*	• It enables to benefit from the good characteristics of predictive and agile processes.	• It could result in an inefficient approach if the advantages/disadvantages of both processes are not well known.
GL2 *Review process*	• It permits to clearly define the review process to speed up the development.	• Misidentification of activities if inexperienced people are in charge of the review process.
GL3 *Documentation templates*	• Precise templates permit to speed up the development while guaranteeing quality. • Reusing templates allows saving time and building on consolidated experience.	• Too specialized templates may not include all the information required by certification standards. • Produce precise template is time-expensive.

GL4 *Resources initial estimation*	• Despite the emergency, investing some time in this upfront activity in the startup phase permits to speed-up the development process, as well as reduce risks.	• Sometimes the competencies and resources needed for a project may be unknown when it is in its initial stage, or their estimation may be not completely reliable.
GL5 *Coordination team and plan*	• This permits to have a coordination team ready to work and have clear indications for the development teams.	• Sometimes the project can be not well defined at the beginning, and thus it can be difficult to define upfront who to insert into the coordination team.
GL6 *Responsibilities assignment*	• Having the coordination team as stable as possible permits avoiding delays that can easily propagate to development teams.	• Some new tasks and responsibility may emerge during the development.
GL7 *Flexible development teams*	• When development teams work agile, they are ready to deal with frequent and unavoidable changes.	• Moving a developer from one task to another can be difficult, if the developer is not familiar with the new one.
GL8 *Inter-team coordination and communication*	• Having clear and stable instruments, tools, and protocols for communication and coordination among different teams permits to easily communicate and focus on development activities.	• The chosen tools may be not the optimal ones for all the needs or users.
GL9 *Intra-team coordination and communication*	• Allowing development teams to select the development, communication, and coordination instruments for the intra-team work permits them to work in their comfort zone.	• Some certification standards require the use of only certified tools during all activities of the software life cycle. In these cases, only a limited set of certified tools should be chosen. • It can be difficult to retrieve the information at a later time if no specific tools are used. • The freedom to choose the instruments is beneficial in the short term but can be chaotic in the long term and in big companies.

Table 9: Benefits and risks of the guidelines on the development process.

Guideline	Benefits	Risks
GL10 *Define a traceability system*	• It is important to properly manage traceability from the initial phases of the project.	• The traceability system might require tuning and adaptation during the entire development. • When the traceability system is not properly defined, some people will not follow it, and it will become useless.
GL11 *Use visual and graphical notations*	• Graphical notations might facilitate the communication among team members with different background and knowledge.	• Focus modeling on the most important parts, otherwise the return on investment may be questionable.
GL12 *Precise requirements and reverse engineering*	• Requirements are important to guide the development and validation phase.	• Writing precise requirements may require a lot of time for complex systems. • If people not experienced in the field are involved, writing precise requirements upfront could be difficult.
GL13 *Define an architecture upfront*	• A clear identification of components and interfaces might help development teams to work independently and in parallel.	• It can be difficult to define all the components upfront since sometimes the need for a new component emerges during the development.
GL14 *Limit the upfront architecture to stable decisions*	• An architecture description can be used to create a common understanding across sites while preserving each team's identity.	• The upfront architecture can become obsolete and misaligned with the implementation.
GL15 *Update the architecture*	• This enables to enrich the upfront architecture with aspects emerging during development and update the architecture accordingly.	• Development team members should interact with software architects, as some emerging aspects may be not seen by them.
GL16 *Exploit communities of practices*	• Community of practices (CoP) enables architects to reason about changes and to reduce assumptions that can become inconsistencies.	• When the CoP is not clearly connected to the management team, it will become less effective and will only play the role of knowledge dissemination.
GL17 *Isolate safety-critical parts*	• The isolation of safety-critical features in specific components or modules permits to limit the validation and certification activities.	• It could result in creating a single point of failure, which should be avoided.

GL18 *Use state machines in specifications*	• State machines are used in various domains and are a good instrument to easily communicate complex behaviors.	• In general, only limited parts of a system can be modeled using state machines. • Consider the artifact as a living object, since it will require to be changed during the development.
GL19 *Use state machines for code generation*	• The use of executable state machines promotes modifiability, maintainability, and understandability.	• Some certification standards may require the use of certified tools for developers, and generating code from state machines can be inapplicable in this case.
GL20 *Different programming languages*	• When the use of different programming languages does not create integration problems, e.g., when the code is deployed on different hardware components, it would be beneficial to allow developers to use familiar languages. • It allows using a language that fits better with the specific needs (e.g, to avoid using C for GUI programming).	• Some certification standard requires the certification of the compiler to be used by developers. In this case, using different programming languages should be avoided. • If coding guidelines are followed, one should assure that they are available for all the chosen programming languages. • Using various programming languages might also make code review activities more difficult.
GL21 *Coding standards and guidelines*	• Coding standards are often required by safety standards and, in general, promote the quality of the code.	• Following coding standards may slow-up the development process if developers are not used to them.
GL22 *Continuous integration and unit testing*	• CI tools and automatic testing instruments enable various teams to work in parallel without breaking the code, and permit to avoid the big bang integration problem.	• For complex systems it may be difficult to initially set up the continuous integration environment.
GL23 *Focus on testing activities*	• Safety-critical components are those that require major attention. Parts of the system that are not critical can follow classic quality management recommendations.	• The usability of a system is affected also by non-safety-critical components, thus focusing only on the critical ones may reduce it.

GL24 *Role of emulators*	• Simulators and emulators can speed up development and validation, but, integration, system, and acceptance testing are unavoidable.	• Simulators may be slightly different from the real environment (e.g., noises and interferences can be difficult to be simulated) and testing using them can be not completely reliable.

Table 10: Benefits and risks of the guidelines on the development phases.

3.5 Conclusions

In this chapter, I presented the reengineering process me and my research team needed to apply in order to make the MVM safe and fitting to the requirements of the medical software certification standards (see Sect. 1.2 for more details). This process led us to derive some lessons learned that can be useful and should be considered before starting the development of a medical device. To make these lessons learned more applicable, this chapter also presents some more prescriptive guidelines, which are given for medical systems, but can be easily extended to all the safety-critical software.
The most important take-away message from this experience is that injecting all the activities needed to certify a medical device after it has already been produced is very difficult and time-consuming. In fact, very often the certification effort is not limited only to paper work but requires refactoring and reengineering the device in a complex way. Thus, developing the medical software in a way coherent with the certification standards from the beginning is always the best choice. Therefore, in the next chapters, I will present methods (mainly model-based) that can be used from the beginning in order to comply with the requirements of certification standards and to produce documentation during the development process.

Part II

Model-based systems engineering for PEMS

Chapter 4. Abstract State Machines for MBSE

In Chapter 3, I have presented the lessons me and my research team learned during the development of a real medical device, namely the MVM. From those lessons learned, we have derived a set of guidelines to be considered for speeding-up the development process, while still remaining compliant to the standards required for medical software certification.

In the guidelines, I have highlighted the importance of structuring the development process in a way that traceability is assured, software requirements are specified before starting the actual software implementation, and formal notations (such as state machines) are used. For this reason, in this chapter, the use of Model-Based System Engineering (MBSE) and, in particular, of Abstract State Machines are presented for dealing with the development and the safety assurance process of PEMS. MBSE methods allow developers to guarantee a defined level of quality, since verification and validation activities are straightforward, and, moreover, using a model-based approach allows one to produce documentation (useful for the certification of medical devices – see Sect. 1.2) in an easier manner. The work presented in this chapter is based on [7] and [30] and is structured as follows. Sect. 4.1 introduces the use of MBSE techniques for PEMS. Then, Sect. 4.2 presents the ASMETA framework, based on Abstract State Machines, which can be used for performing MBSE activities during the development process of PEMS. The use and functionalities of the framework during the design time are presented in Sect. 4.3, while Sect. 4.4 explains how ASMETA can be applied at development-time for automatically generating code from models. Finally, Sect. 4.5 presents the use of the ASMETA framework for runtime simulation at operation time, and Sect. 4.6 concludes the chapter.

4.1 Introduction

Failures of safety-critical systems, such as medical devices, could have potentially large and catastrophic consequences, such as human hazards or even loss of human life (see Sect. 1.1.1 for some example) [119]. To ensure safe operation and avoid dangerous consequences of system failure, medical safety-critical systems need development methods and processes that lead to provably correct systems. Rigorous development processes require the use, for example, of formal methods, which can guarantee, thanks to their mathematical foundation, model preciseness, and properties assurance.

A particular aspect of medical devices is that the use of formal methods can be more challenging since "system safety" is not only "software safety" but may depend on the use of the software within its untrusted and unreliable environment, which may include a possible patient. However, even in

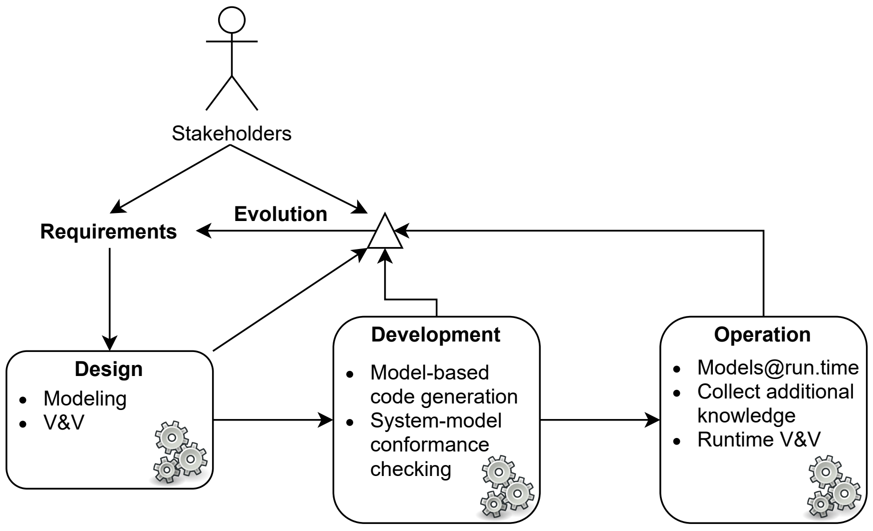


Figure 18: Safety assurance MBSE process during system's life cycle.

these cases, developers and testers should be sure that the developed systems comply with their specifications. Since this process may be difficult to be carried out, automatic techniques based on formal models are classically used by developers.

This approach is commonly referred to in the literature as *Model-based systems engineering* (MBSE), i.e., the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities, beginning in the conceptual design phase (modeling) and continuing throughout the development and later phases of the life cycle. Fig. 18 outlines this process, showing the three main phases of the Design, Development, and Operation of a system life cycle. In particular, the following phases are identified:

- During the *design* phase, systems models are created, verified, and validated;
- During the *system development* phase, models, which have been already validated and verified, are eventually used to derive correct-by-construction code/artifacts of the system and/or to check that the developed system conforms to its models;
- During the *operation* phase, models introduced at design-time are optionally executed in tandem with the system to perform analysis at runtime.

Throughout this assurance process, stakeholders and system developers jointly derive and integrate new evidence and arguments for analysis (Δ); system requirements and models can eventually be adapted according to the knowledge collected. Hence, requirements and models evolve accordingly throughout the system life cycle.

This safety assurance process requires the availability of formal approaches with specific characteristics in order to cover all the three identified phases: models should be possibly executable for high-level design validation and enriched with properties verification mechanisms; models should be operational-based in order to support easily code generation from them and model-based testing activities; state-based methods are suitable for co-simulation between model and code and for checking state conformance between model state and code state at runtime. In principle, different methods and tools can be used in all the three phases; however, the integrated use of different tools around the same formal method is much more convenient than having different tools working on input models with their own languages and writing several translators. For this reason, this chapter and all the experiments on MBSE presented in this book are entirely based on the ASMETA[1] framework, which is introduced and detailed in the following sections.

4.2 The ASMETA framework

This section recalls the origin of the ASMETA project [17] and the basic concepts of the ASM method on which it is based. The tools composing the ASMETA framework will also be presented under the light of the safety assurance process.

The ASMETA project started in 2004 with the goal of overcoming the lack of tools supporting the ASM formalism. At that time, the formal approach had already shown to be widely used for the specification and verification of a number of software systems and in different application domains [44]; however, ASMs were not considered suitable for practical use, since there was a lack of tools supporting them. The main goal of the ASMETA project was to develop a textual notation for encoding ASM models, by exploiting the Model-driven Engineering (MDE) approach [157], to develop an abstract syntax of a modeling language for ASMs [90] in terms of a metamodel, and to derive from it a user-facing textual notation to edit ASM models. Then, from the ASM metamodel and by exploiting the runtime support for models and model transformation facilities of the open-source Eclipse-based environment, ASMETA has been progressively developed till now as an Eclipse-based set of tools for ASM model editing, visualization, simulation, validation, property verification, and model-based testing [13].

In order to support a variety of analysis activities on ASM models, ASMETA is integrated with different external tools, such as the NuSMV[2] model checker for performing property verification, and SMT solvers (e.g., Yices[3]) to support correct model refinement verification and runtime verification. For this purpose, ASMETA mainly supports a black-box model composition strategy based on semantic mapping, i.e., ASM models are automatically transformed to be compatible with the input formalism

[1]`ttps://asmeta.github.io/`

[2]`https://nusmv.fbk.eu/`

[3]`https://yices.csl.sri.com/`

of the target tool, the analysis is performed by the tool, and then the analysis results are lifted back to the ASM level.

4.2.1 Background concepts for Abstract State Machines

The computational model at the base of the ASMETA framework is that of the Abstract State Machines (ASMs) formal method, which was originally introduced by Yuri Gurevich as *Evolving Algebras* [96]. ASMs are mainly based on two concepts:

- ASM *states*, which replace unstructured FSM control states with algebraic structures, i.e., domains of objects, together with functions and predicates defined on them.
- ASM *location*, defined as a pair (*function-name*, *list-of-possible-parameter-values*), which represents the ASM concept of the basic object container, and the couple (*location*, *value*) is a memory unit.

Note that, given the two definitions above, an ASM state can be viewed as a set of abstract memory units.

State transitions are performed by firing *transition rules*, which represents a modification of the interpretation of functions from one state to the next, and therefore they change the value of specific locations. Location *updates* are given as assignments of the form *loc* := *v*, where loc is a location and *v* its new value. They are the basic units of rules construction and can be combined for building more complex rules. By a limited but powerful set of *rule constructors*, location updates can be combined to express other forms of machine actions as: guarded/conditional actions (`if-then`, `switch-case`), simultaneous parallel actions (`par` and `forall`), sequential actions (`seq`), and non-deterministic actions (`choose`).

Functions that are not updated by rule transitions are considered as *static*. On the other hand, the functions that are updated are defined as *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and its environment).

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is the initial state and each S_{n+1} is obtained from S_n by firing the set of all transition rules invoked by a unique *main rule*, which can be seen as the starting point of the computation.

It is also possible to specify state *invariants* as first-order logic formulas that must be true in each computational state. A set of safety assertions can be specified as model invariants, and a model state is *safe* if state invariants are always satisfied.

ASMs allow modeling different computational paradigms, from a *single* agent to distributed *multiple* agents. In particular, a *multi-agent ASM* is a family of pairs $(a, ASM(a))$, where each a of a predefined set *Agent* executes its own machine $ASM(a)$, specifying the agent's behavior, and contributes to determine the next state by interacting synchronously or asynchronously with the other agents.

ASMs offer several advantages over other automata-based formalisms: (1) due to their *pseudo-code format*, they can be easily understood by practitioners and can be used for high-level programming; (2) they offer a precise system specification at any desired *level of abstraction*; (3) they are *executable models*, so they can be co-executed with system low-level implementations [153]; (4) *model refinement* (see Sect. 4.3.1 for more details) is an embedded concept in the ASM formal approach; it allows for facing the complexity of system specifications by starting with a high-level description and then proceeding step-by-step by adding further details till a desired level complexity and completeness has been reached; (5) ASMs support the concept of ASM *modularization*, i.e., they allow for defining an ASM without the main firing rule, which facilitates model scalability and separation of concerns, so tackling the complexity of big systems specification; (6) they support synch/async multi-agent compositions, which allows for *modeling distributed and decentralized software systems* [15].

4.2.2 ASMETA toolset

Fig. 19 gives an overview of the ASMETA tools by showing their use to support the different activities of the safety assurance process, e.g., for medical devices.

At *design-time*, ASMETA allows for using a number of tools for model editing and visualization (the modeling language `AsmetaL`, its editor `AsmetaXt` and compiler `AsmetaC`, and the model visualizer `AsmetaVis` for graphical visualization of ASM models), model validation (e.g., interactive or random simulation by the simulator `AsmetaS`, animation by the animator `AsmetaA`, scenario construction using the `Avalla` language, and validation by the validator `AsmetaV`), and verification (e.g., static analysis by the model reviewer `AsmetaMA`, proof of temporal properties by the model checker `AsmetaSMV`, proof of correct model refinement process by `AsmRefProver`).

At *development-time*, ASMETA supports automatic code and test case generation from models (the code generator `Asm2C++`, the unit test generator `ATGT`, and the acceptance test generator `AsmetaBDD` for complex system scenarios). The code automatically generated can be used in a variety of embedded systems (e.g., Arduino).

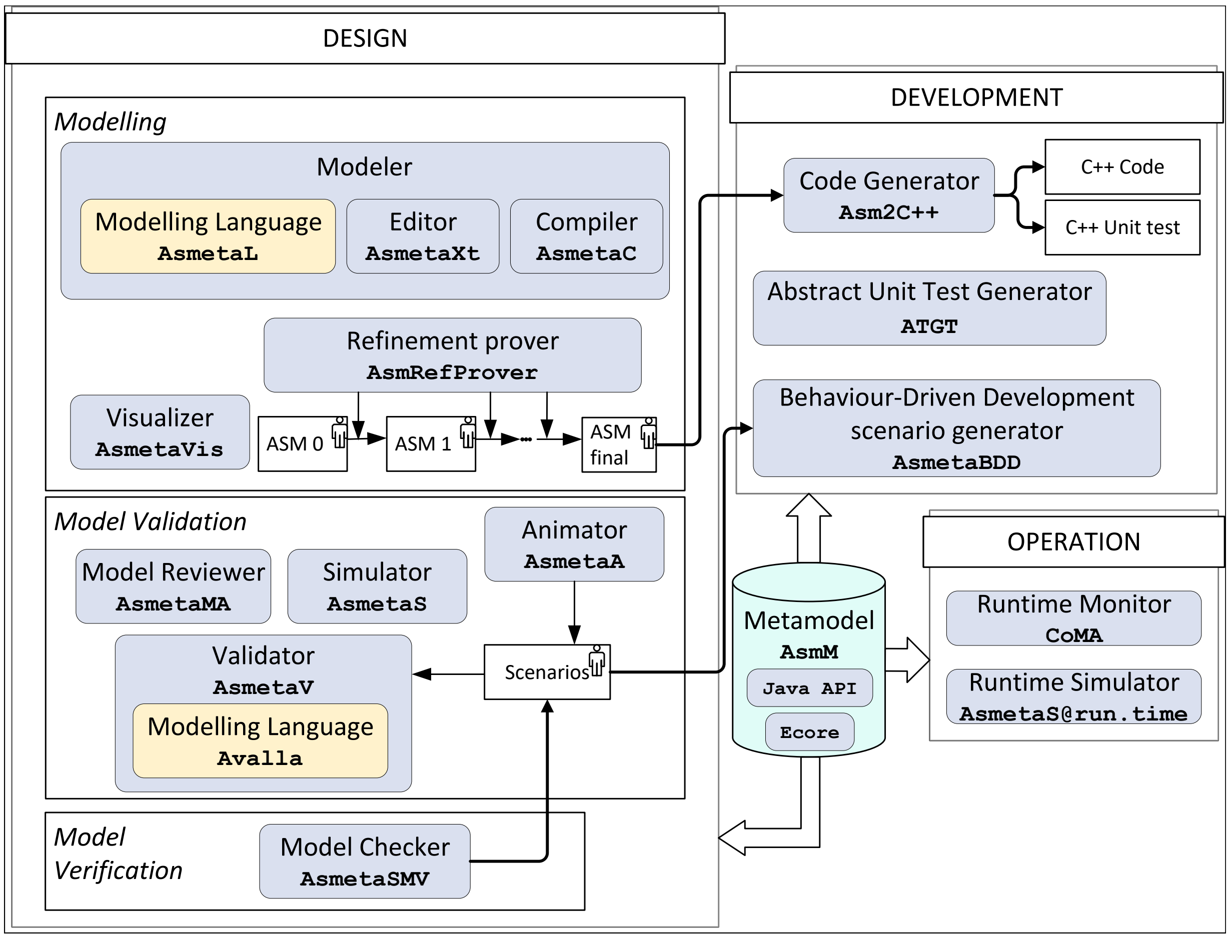


Figure 19: ASMETA tool-set.

Finally, at *operation-time*, ASMETA supports runtime simulation (with the simulator `AsmetaS@run.time`) and runtime monitoring (using the tool `CoMA`).

The analysis techniques and associated tooling strategies supported by ASMETA are described in more detail in the next sections.

4.3 ASMETA @ design-time

System design is the first activity of the MBSE process supported by ASMETA. During this phase, users can model the desired system using the `AsmetaL` language and refine every model, which can be visualized graphically and analyzed with several tools for verification and validation.

4.3.1 Modeling

Starting from the functional requirements, ASMETA allows the user to model the system by using, if needed, model composition and refinement. Furthermore, models can be visualized through the `AsmetaVis` tool. As an example, the MVM case study (see Chapter 2 for more details) is used to show the ASMETA functionalities in the following.

Modeling language

System requirements are modeled in ASMETA using the `AsmetaL` language and the `AsmetaXt` editor. Listing 4.1 shows a preliminary example of the abstract `AsmetaL` model of the MVM, in which only the PCV mode is implemented (see Chapter 2 for further details on ventilation modes). The model, identified by a name after the keyword `asm`, is structured into four sections:

- The *header*, where the signature (functions and domains) is declared, and the external signature is imported (see the description of the modularization technique below);
- The *body*, where transitions rules, possible concrete domains and derived functions are defined;
- A *main rule*, which defines the starting point of the machine;
- The *initialization*, where a default initial state (among a set of possibile states) is defined.

As previously described in Sect. 4.1, each `AsmetaL` rule can be defined by using a set of rule constructors to express the different machine action paradigms.

Modularization

ASMETA supports the mechanisms of modularization and information-hiding, by exploiting the module notation. In fact, when requirements are complex or when separation of concerns is desired, users can organize the model in several ASM modules and join them, by using the import statement (see Listing 4.1), into a single main one (also defined as *machine*). The main asm is declared as `asm`, it imports the other modules and may access to functions, rules, and domains declared within the sub-modules. Indeed, every ASM *module* may contain definitions of domains, functions, invariants, and rules, while the ASM machine is a module that additionally contains an initial state and the main rule, representing the starting point of the execution.

Modeling time with ASMETA

From our experience, of which the work presented in this book is an important part, ASMETA can be effectively used for modeling real-world PEMS. However, many real systems, especially those in the safety-critical and medical domains, rely on time constraints. For this reason, in the ASMETA framework, the TimeLibrary[4] (see Listing 4.2) has been introduced. It contains the basic constructs necessary to handle time features in ASMETA specifications: *i)* monitored functions to manage the time in different time units (nanoseconds, milliseconds, seconds, minutes, and hours); *ii)* an abstract domain Timer useful to introduce user-defined timers; *iii)* some functions and rules to operate on

[4]`https://github.com/asmeta/asmeta/blob/master/asm_examples/STDL/TimeLibrary.asm`

```
asm MVM0
import StandardLibrary

signature:
  // TIMER
  enum domain Timer = {TIMER_INSPIRATION_DURATION_MS,
        TIMER_EXPIRATION_DURATION_MS,
        TIMER_TRIGGERWINDOWDELAY_MS}
  domain MyTime subsetof Integer
  // DOMAINS
  enum domain States = {STARTUP, SELFTEST,
        VENTILATIONOFF, PCV_INSPIRATION, PCV_EXPIRATION,
        OFF}
  enum domain Modes = {PCV}
  enum domain ValveStatus = {OPEN , CLOSED}
  // FUNCTIONS
  dynamic monitored poweroff: Boolean
  dynamic monitored startupEnded: Boolean
  dynamic monitored selfTestPassed: Boolean
  dynamic monitored resume: Boolean
  dynamic monitored startVentilation: Boolean
  dynamic monitored stopVentilation: Boolean
  dynamic monitored mode: Modes
  dynamic monitored pawGTMaxPinsp: Boolean
  dynamic monitored dropPAW_ITS_PCV: Boolean
  dynamic controlled time: MyTime
  dynamic controlled stopVentilationRequested: Boolean
  dynamic controlled state: States
  dynamic controlled iValve: ValveStatus
  dynamic controlled oValve: ValveStatus
  controlled start: Timer-> MyTime
  static durationTIMER_INSPIRATION_DURATION_MS : MyTime
  static durationTIMER_EXPIRATION_DURATION_MS: MyTime
  static durationTIMER_TRIGGERWINDOWDELAY_MS : MyTime
  derived expiredTIMER_INSPIRATION_DURATION_MS : Boolean
  derived expiredTIMER_EXPIRATION_DURATION_MS: Boolean
  derived expiredTIMER_TRIGGERWINDOWDELAY_MS : Boolean

definitions:
  domain MyTime = {0:600}
  function durationTIMER_INSPIRATION_DURATION_MS  = 20
  function durationTIMER_EXPIRATION_DURATION_MS = 40
  function durationTIMER_TRIGGERWINDOWDELAY_MS = 10
  function expiredTIMER_INSPIRATION_DURATION_MS = (time >=
        start(TIMER_INSPIRATION_DURATION_MS) +
        durationTIMER_INSPIRATION_DURATION_MS)
  function expiredTIMER_EXPIRATION_DURATION_MS = (time >=
        start(TIMER_EXPIRATION_DURATION_MS) +
        durationTIMER_EXPIRATION_DURATION_MS)
  function expiredTIMER_TRIGGERWINDOWDELAY_MS = (time >=
         start(TIMER_TRIGGERWINDOWDELAY_MS) +
        durationTIMER_TRIGGERWINDOWDELAY_MS)
  macro rule r_reset_TIMER($t in Timer) = start($t) := time

  // RULE DEFINITIONS
  rule r_startupEnded = state := SELFTEST
  rule r_ventOffRequested = stopVentilationRequested := true
  rule r_ventOffPCV = par
      state := VENTILATIONOFF
      stopVentilationRequested := false
    endpar
  rule r_ventOffFT = state := VENTILATIONOFF
  rule r_turnOff = par
      iValve := CLOSED
      oValve := OPEN
      state := OFF
    endpar
  rule r_PCVinsp = par
      state := PCV_INSPIRATION
      iValve := OPEN
      r_reset_TIMER[TIMER_INSPIRATION_DURATION_MS]
    endpar
  rule r_PCVinspOValve = par
      r_PCVinsp[]
      oValve := CLOSED
    endpar
  rule r_PCVexp = par
      state := PCV_EXPIRATION
      oValve := OPEN
      r_reset_TIMER[TIMER_EXPIRATION_DURATION_MS]
      r_reset_TIMER[TIMER_TRIGGERWINDOWDELAY_MS]
    endpar
  rule r_PCVexpIValve = par
      r_PCVexp[]
      iValve := CLOSED
    endpar

  // MAIN Rule
  main rule r_Main = par
      time:= time+1
      if poweroff then r_turnOff[]
      //if stop ventilation is requested and current state is expiration
      //go to state VENTILATIONOFF immediately
      else
        if state=PCV_EXPIRATION and (stopVentilationRequested or
              stopVentilation) then
          r_ventOffPCV[]
        else par
          //if ventilation stop is requested and ventilation is on, store
          //the stop request in the function stopVentilationReq.
          if stopVentilation then
            if state!=PCV_EXPIRATION and state!=STARTUP and
                state!=SELFTEST and state!=VENTILATIONOFF
                then r_ventOffRequested[] endif endif
          //transition from startup to selftest
          if state = STARTUP then
            if startupEnded then r_startupEnded[] endif endif
          //transition from selftest to ventilation off
          if state = SELFTEST then
            if (selfTestPassed or resume) then r_ventOffFT[] endif
                endif
          //start ventilation, in PCV mode
          if state = VENTILATIONOFF then
            if startVentilation then
              if mode = PCV then r_PCVinspOValve[] endif endif
                endif
          //transition from inspiration to expiration
          if state = PCV_INSPIRATION then
            if expiredTIMER_INSPIRATION_DURATION_MS then
              if mode = PCV then r_PCVexpIValve[] endif
            else if pawGTMaxPinsp then r_PCVexpIValve[] endif
                endif endif
          if state = PCV_EXPIRATION then
            if expiredTIMER_EXPIRATION_DURATION_MS then
              r_PCVinspOValve[]
            else
              if expiredTIMER_TRIGGERWINDOWDELAY_MS then if
                dropPAW_ITS_PCV then r_PCVinspOValve[]
                endif endif endif endif endpar endif endif
                endpar

default init s0:
  function time = 0
  function state = STARTUP
  function iValve = CLOSED
  function oValve = OPEN
  function stopVentilationRequested = false
  function start($t in Timer) = 0
```

Listing 4.1: AsmetaL specification for the MVM case study.

```
module TimeLibrary
import StandardLibrary
export *
signature:
  abstract domain Timer
  enum domain TimerUnit={NANOSEC,
        MILLISEC, SEC, MIN, HOUR}
  monitored mCurrTimeNanosecs: Integer
  monitored mCurrTimeMillisecs: Integer
  monitored mCurrTimeSecs: Integer
  monitored mCurrTimeMins: Integer
  monitored mCurrTimeHours: Integer
  controlled start: Timer-> Integer
  controlled duration: Timer -> Integer
  controlled timerUnit: Timer -> TimerUnit
  derived currentTime : Timer-> Integer
  derived expired: Timer -> Boolean
```

```
definitions:

  function currentTime($t in Timer) = if (timerUnit($t)=NANOSEC) then
    mCurrTimeNanosecs
    else if (timerUnit($t)=MILLISEC) then mCurrTimeMillisecs
    else if (timerUnit($t)=SEC) then mCurrTimeSecs
    else if (timerUnit($t)=MIN) then mCurrTimeMins
    else if (timerUnit($t)=HOUR) then mCurrTimeHours
    endif endif endif endif endif
  function expired($t in Timer) = (currentTime($t) >= start($t) + duration($t))

  macro rule r_reset_timer($t in Timer) = start($t) :=
    currentTime($t)
  macro rule r_set_duration($t in Timer, $ms in Integer) =
    duration($t) := $ms
  macro rule r_set_timer_unit($t in Timer, $unit in TimerUnit) =
    timerUnit($t) := $unit
```

Listing 4.2: ASMETA TimeLibrary.

timers, like checking if a desired amount of time is passed, resetting and starting a timer, and setting the timer duration and time unit. The implemented solution allows users to use different time units in the same ASM specification, and it guarantees consistency between them during model simulation. Moreover, this mechanism ensures that in a defined state, all time functions refer to the same time instant, no matter what time unit is used. A simple example using the time monitored functions is shown in Listing 4.3, representing a clock displaying at each step the current hours, minutes, and seconds.

```
asm simpleClock
import TimeLibrary

signature:
  controlled clockHours: Integer
  controlled clockMins: Integer
  controlled clockSecs: Integer
```

```
definitions:
  main rule r_main =
    par
      clockHours:=mCurrTimeHours mod 24
      clockMins:=mCurrTimeMins mod 60
      clockSecs:=mCurrTimeSecs mod 60
    endpar
```

Listing 4.3: Time example - return current time.

Measuring the absolute time is useful but, often, systems require that actions are executed if a desired amount of time is passed. For this purpose, timers are available in the TimeLibrary, too. After having declared a timer, users can reset it through the rule `r_reset_timer`, change its duration with the rule `r_set_duration`, or its time unit using the rule `r_set_timer_unit`.

Refinement

The modeling process of an ASM is usually based on model refinement [42]: the designer starts with a high-level description of the system and proceeds through a sequence of more detailed models, each introducing, step-by-step, design decisions and details. In order to adopt a correct refinement process, at each refinement level, the model must be proved to be a correct refinement of the more abstract one.

ASMETA supports a special case of *1-n refinement*, consisting in adding functions and rules in a way that one step in the ASM at a higher level can be performed by several steps in the refined model. Indeed, a refinement is considered *correct* if any behavior (i.e., run or sequence of states) in the refined model can be mapped to a run in the abstract model. To automatically prove the correctness of the model refinement process, users can exploit the `AsmRefProver` tool [12], which is based on a Satisfiability Modulo Theories (SMT) solver. When executing this tool, one can specify two refinement levels and ensure that an ASM specification ASM_i is a correct refinement of a more abstract one ASM_{i-1}. Then, `AsmRefProver` confirms whether the refinement is correctly performed with two different outputs: `Initial states are conformant` and `Generic step is conformant`.

Listing 4.4 shows an excerpt of the refinement for the MVM case study (see Listing 4.1) in which the PSV mode is introduced. Thus, the behavior of the system modeled in Listing 4.1 is preserved and expanded during the refinement process.

Modeling by refinement allows users to add to the model additional requirements of increasing complexity only when the developer has gained enough confidence in the basic behaviors of the modeled system. This can be done by alternating modeling and testing activities, as presented in [28], with different refinement levels.

Visualization

Model visualization is a good way for people to communicate and get a common understanding of the modeled system. ASMETA supports model visualization by a visual notation defined in terms of a set of construction rules and schema that give a graphical representation of an ASM and its rules [9]. The graphical information is represented in a visual graph in which nodes represent syntactic elements (like rules, conditions, rule invocations) or states, while edges represent bindings between syntactic elements or state transitions. The `AsmetaVis` tool can perform two types of visualization: *basic visualization*, which shows the syntactic structure of the model and returns a visual tree obtained by recursively visiting the ASM rules; *semantic visualization*, which exploits visual patterns that permit to capture some behavioral information as control states.

4.3.2 Validation and verification

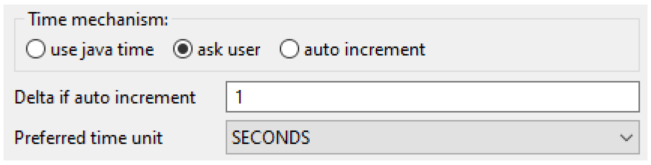
Once the `AsmetaL` model is available, the user can take advantage of the tools offered by the ASMETA framework to perform validation and verification activities.

```
asm MVM1

import StandardLibrary
import CTLlibrary
import LTLlibrary

signature:
  // TIMER
  enum domain Timer = {TIMER_INSPIRATION_DURATION_MS,
        TIMER_EXPIRATION_DURATION_MS,
        TIMER_MAX_INSP_TIME_PSV, TIMER_MIN_EXP_TIME_PSV,
        TIMER_TRIGGERWINDOWDELAY_MS,
        TIMER_MIN_INSP_TIME_MS}
  domain MyTime subsetof Integer

  // DOMAINS
  enum domain States = {STARTUP, SELFTEST, VENTILATIONOFF,
        PCV_INSPIRATION, PCV_EXPIRATION, PSV_INSPIRATION,
        PSV_EXPIRATION, OFF}
  enum domain Modes = {PCV, PSV}
  enum domain ValveStatus = {OPEN, CLOSED}

  // FUNCTIONS
  dynamic monitored flowDropPSV: Boolean
  dynamic monitored dropPAW_ITS_PSV: Boolean
  [...]
  static durationTIMER_MAX_INSP_TIME_PSV: MyTime
  static durationTIMER_MIN_EXP_TIME_PSV: MyTime
  static durationTIMER_MIN_INSP_TIME_MS: MyTime
  derived expiredTIMER_MAX_INSP_TIME_PSV: Boolean
  derived expiredTIMER_MIN_EXP_TIME_PSV: Boolean
  derived expiredTIMER_MIN_INSP_TIME_MS: Boolean

definitions:

  [...]

  rule r_ventOffPSV =
    par
      state := VENTILATIONOFF
      stopVentilationRequested := false
    endpar

  rule r_PSVinsp = par
      state := PSV_INSPIRATION
      iValve := OPEN
      r_reset_TIMER[TIMER_MAX_INSP_TIME_PSV]
      r_reset_TIMER[TIMER_MIN_INSP_TIME_MS]
    endpar

  rule r_PSVinspOValve =
    par
      r_PSVinsp[]
      oValve := CLOSED
      r_reset_TIMER[TIMER_MAX_INSP_TIME_PSV]
    endpar

  rule r_PSVexp =
    par
      state := PSV_EXPIRATION
      oValve := OPEN
      r_reset_TIMER[TIMER_MIN_EXP_TIME_PSV]
      r_reset_TIMER[TIMER_TRIGGERWINDOWDELAY_MS]
    endpar

  rule r_PSVexpIValve =
    par
      r_PSVexp[]
      iValve := CLOSED
    endpar

  rule r_PSVexpIValveFromPCV =
    par
      r_PSVexp[]
      iValve := CLOSED
    endpar

  // MAIN Rule
  main rule r_Main =
      [...]

default init s0:
  function time = 0
  function state = STARTUP
  function iValve = CLOSED
  function oValve = OPEN
  function stopVentilationRequested = false
  function start($t in Timer) = 0
```

Listing 4.4: Example of a refined AsmetaL specification for the MVM.

Simulation

Simulation is the first validation activity that allows for checking the AsmetaL model behavior during its development, and it is supported by the AsmetaS tool [14]. Given a model, at every step, the simulator computes the update set based on the theoretical definitions given in [44] and constructs the model run. The simulator supports two types of simulation: *random* and *interactive*. In the former, it automatically assigns values to monitored functions, randomly choosing their values from their codomains. In the latter, instead, the user is requested to insert the value of monitored functions and, in case of input errors, a message invites the user to insert again the function value. Similarly, in the case of invariant violation or inconsistent updates, the simulation is interrupted, and an error message is shown in the console.


Time mechanism:
use java time [✓ ask user] auto increment

Delta if auto increment: 1

Preferred time unit: SECONDS

Figure 20: Simulator settings in Eclipse preferences.

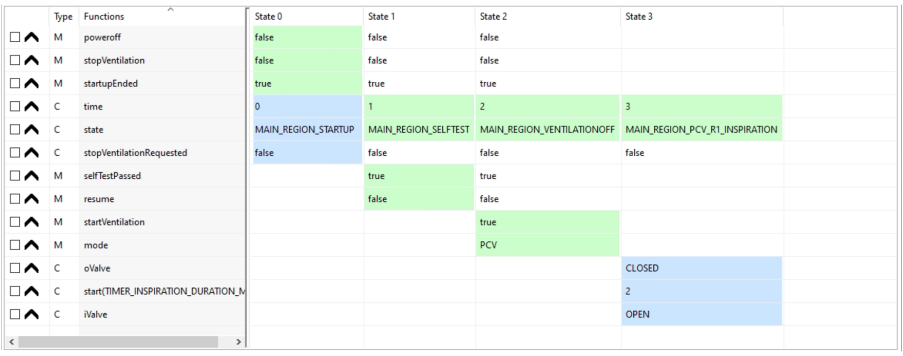
Simulation of ASMETA specification with time features ASMETA supports three different mechanisms to handle time during simulation if the `TimeLibrary` is used (see Sect. 4.3.1):

1. *Use java time*: the time is read from the machine hosting the simulation;
2. *Ask user*: the user sets the value for the time at each step as normal monitored functions;
3. *Auto increment*: The time is automatically increased at each step by a predefined value.

The first mechanism allows the user to run the specification without inserting the value of the functions representing the time because their value is obtained from the Java 8 Date/Time API `Instant.now()` and automatically assigned to them. However, especially if the specification requires long time intervals, like hours, or very short time intervals, like nanoseconds, if the real time is used during the simulation, it may be unfeasible for the user to check what happens at specific instants of time. In this case, the second mechanism is most suitable: the user specifies the time unit to which he wants to run the specification and inserts the desired time value when required. Note that if the specification uses more than one time function with different time units, the others are automatically derived starting from the one inserted by the user. Finally, in case the user wants to execute the specification and automatically increment the time by a predefined value *delta* at each step, the third approach can be used. The user has to define the time step and time unit; then the system automatically increments the time of the chosen delta value at each running step. Even in this case, if time functions have other time units compared to the one set by the user, they are automatically derived. The desired mechanism can be set in the ASMETA → Simulator preferences from the Window menu in Eclipse, as shown in Fig. 20.

Animation

The main disadvantage of the simulator is that its interface is only textual, and this sometimes makes it difficult to follow the model computation. For this reason, ASMETA embeds a model animator, `AsmetaA` [36], providing the user with complete information about all locations and using colors, tables, and figures over simple text to convey information about states and their evolution. This tool helps the user to follow the model computation and understand how the system state changes at every step.



Type	Functions	State 0	State 1	State 2	State 3
M	poweroff	false	false	false	
M	stopVentilation	false	false	false	
M	startupEnded	true	true	true	
C	time	0	1	2	3
C	state	MAIN_REGION_STARTUP	MAIN_REGION_SELFTEST	MAIN_REGION_VENTILATIONOFF	MAIN_REGION_PCV_R1_INSPIRATION
C	stopVentilationRequested	false	false	false	false
M	selfTestPassed		true	true	
M	resume		false	false	
M	startVentilation			true	
M	mode			PCV	
C	oValve				CLOSED
C	start(TIMER_INSPIRATION_DURATION_M				2
C	iValve				OPEN

Figure 21: Animation of the MVM using `AsmetaA`.

Since the animator runs the simulator in the background, it also supports *interactive* and *random* animation. In the interactive mode, the insertion of input functions is pursued through different dialog boxes depending on the type of function to be inserted. If the function value is not in its codomain, the animator keeps asking until an acceptable value is inserted. In random mode, the monitored function values are automatically and randomly assigned.

With complex models, running one random step at a time may be tedious; for this reason, the user can also specify the number of steps to be performed, and the tool performs the random simulation accordingly. In case of invariant violation, or an inconsistent update, a message is shown in a dedicated text box and the animation is interrupted. Once the user has animated the model, the tool allows exporting the model run as a scenario, so that it can be re-executed whenever desired. Fig. 21 shows the animation of the initial steps of the MVM.

Scenario-based validation

Both `AsmetaS` and `AsmetaA` tools require the user to execute the `AsmetaL` model step by step, everytime the model has to be validated. Instead, with scenario-based validation, the user can write a *scenario*, i.e., a description of external actions and reactions of the system [55] that can be executed whenever needed to check the model behavior. Typical uses of scenarios are regression or classical unit testing activities.

The scenarios are written in the `Avalla` language and executed using the `AsmetaV` tool. Each scenario is identified by its name and must `load` the ASM to be tested. Then, the user may specify different commands depending on the operation to be performed:

- The `set` command updates monitored or shared function values that are supplied by the user as input signals to the system;
- Commands `step` and `step until` represent the reaction of the system, which can execute one single ASM step and one ASM step iteratively until a specified condition becomes true;

```
scenario test1
load MVM0.asm

check iValve = CLOSED;
check expiredTIMER_EXPIRATION_DURATION_MS = false;
check start(TIMER_TRIGGERWINDOWDELAY_MS) = 0;
check start(TIMER_EXPIRATION_DURATION_MS) = 0;
check durationTIMER_INSPIRATION_DURATION_MS = 20;
check durationTIMER_EXPIRATION_DURATION_MS = 40;
check time = 0;
check durationTIMER_TRIGGERWINDOWDELAY_MS = 10;
check expiredTIMER_TRIGGERWINDOWDELAY_MS = false;
check stopVentilationRequested = false;
check start(TIMER_INSPIRATION_DURATION_MS) = 0;
check oValve = OPEN;
check state = STARTUP;
check expiredTIMER_INSPIRATION_DURATION_MS = false;
set poweroff := false;
set startupEnded := false;
set stopVentilation := false;
step
check time = 1;
set poweroff := false;
set startupEnded := false;
set stopVentilation := false;
step
```

Listing 4.5: Example of `Avalla` scenario for the MVM case study.

- The `check` command is used to inspect property values in the current state of the underlying ASM.

Listing 4.5 shows an example of the `Avalla` scenario for the MVM case study. To simulate scenarios, `AsmetaV` exploits the simulator. Moreover, during the simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a "PASS" verdict ("FAIL" otherwise). Moreover, the tool collects information about the coverage of the `AsmetaL` model. In particular, it keeps track of all the rules that have been called and evaluated, and it lists them at the end.

Users can exploit modularization even when building scenarios. Indeed, it is possible to define `blocks`, i.e., sequences of `set`, `step`, and `check`, that can be recalled using the `execblock` when writing other scenarios that foresee the same sequence of `Avalla` commands.

Model review

During the definition of a formal model, a developer may introduce some errors that are not related to a wrong specification of the requirements, but are due to carelessness, forgetfulness, or limited knowledge of the formal method. For example, a developer may use a wrong function name, forget to properly guard an update, and so on. A common error in the development of ASM is *inconsistent update*, i.e., when a location is simultaneously updated to two different values by two rules that are executed in parallel [43]. Such kind of error may occur (especially in complex models) because the developer does not properly guard all the updates. Other types of errors done using ASMs are *overspecifying* the model, i.e., adding model elements that are not needed or writing rules that can never be triggered.

These types of errors can be captured automatically by performing a static analysis of the model. This is the aim of the `AsmetaMA` tool [11], which performs *automatic* review of ASM models. It checks the presence of seven types of errors by using suitable *meta-properties* specified in CTL and verified using the model checker `AsmetaSMV` (see the next subsection for details about `AsmetaSMV`). In particular, the following meta-properties are checked:

- P1 No inconsistent update is ever performed;
- P2 Every conditional rule is complete;
- P3 Every rule can eventually fire;
- P4 No assignment is always trivial[5];
- P5 For every domain element *e* there exists a location which has value *e*;
- P6 Every controlled function can take any value in its co-domain;
- P7 Every controlled location is updated and every location is read.

Model checking

ASMETA provides model checking support using the tool `AsmetaSMV` [10] that translates an ASM specification into a model of the symbolic model checker NuSMV [61], which is used to perform the verification. Being NuSMV a finite state model checker, the only limitation of `AsmetaSMV` is the finiteness of the number of ASM states: only finite domains can be used, and adding elements at runtime to a domain is not supported.

When using `AsmetaSMV`, the user does not need to know how to translate the model into the NuSMV format: it is possible to specify directly in the ASM model the Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) properties defined over the ASM signature. Listing 4.6 shows CTL and LTL properties specified for the MVM study. The CTL property, for example, checks that when the ventilation is not active, the output valve is open and the in valve is closed.

In order to better understand the verification results, the tool allows users to simulate the returned counterexample: a translator takes as input the counterexample given by NuSMV when a property is not verified and produces an `Avalla` scenario. Listing 4.7 shows the counterexample of a violation of an LTL property by a faulty version of the ASM specification of the MVM case study; the corresponding `Avalla` scenario is reported in Listing 4.8.

[5] An assignment is trivial in ASM if the location is updated to the value that it already has.

```
// When ventilation is off, out valve is open and in valve is closed
CTLSPEC ag(state=MAIN_REGION_VENTILATIONOFF implies (iValve=CLOSED and oValve=OPEN))

// Once turned off, the state doesn't change anymore
LTLSPEC g(state=OFF implies g(state=OFF))
```

Listing 4.6 Specification of temporal properties in the AsmetaL model.

```
-- specification G iValve = CLOSED is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    iValve = CLOSED
    time = 0
    start(TIMER_INSPIRATION_DURATION_MS) = 0
    state = MAIN_REGION_STARTUP
    poweroff = false
    stopVentilationRequested = false
    stopVentilation = false
    pawGTMaxPinsp = false
    start(TIMER_EXPIRATION_DURATION_MS) = 0
    dropPAW_ITS_PCV = false
    start(TIMER_TRIGGERWINDOWDELAY_MS) = 0
    startVentilation = false
    resume = false
    [...]
    mode = PCV
  -> State: 1.2 <-
    time = 1
    startupEnded = true
    expiredTIMER_TRIGGERWINDOWDELAY_MS = true
  -> State: 1.3 <-
    time = 2
    state = MAIN_REGION_SELFTEST
    selfTestPassed = true
    startupEnded = false
    expiredTIMER_INSPIRATION_DURATION_MS = true
  -> State: 1.4 <-
    time = 3
    state = MAIN_REGION_VENTILATIONOFF
    startVentilation = true
    selfTestPassed = false
  -> State: 1.5 <-
    iValve = OPEN
    time = 4
    start(TIMER_INSPIRATION_DURATION_MS) = 3
    state = MAIN_REGION_PCV_R1_INSPIRATION
    poweroff = true
    startVentilation = false
    expiredTIMER_INSPIRATION_DURATION_MS = false
    expiredTIMER_EXPIRATION_DURATION_MS = true
  -> State: 1.6 <-
    iValve = CLOSED
    time = 5
    state = OFF
    poweroff = false
    expiredTIMER_INSPIRATION_DURATION_MS = true
  -> State: 1.7 <-
    [...]
  -- Loop starts here
  -> State: 1.61 <-
    time = 60
  -> State: 1.62 <-
```

Listing 4.7 Counterexample in AsmetaSMV.

```
scenario mvmtest
load MVM_0.asm

check iValve = CLOSED;
check time = 0;
check start(TIMER_INSPIRATION_DURATION_MS) = 0;
check state = MAIN_REGION_STARTUP;
check stopVentilationRequested = false;
check start(TIMER_EXPIRATION_DURATION_MS) = 0;
check start(TIMER_TRIGGERWINDOWDELAY_MS) = 0;
set startVentilation = false;
set resume := false;
set selfTestPassed := false;
set startupEnded := false;
set expiredTIMER_TRIGGERWINDOWDELAY_MS := false;
set expiredTIMER_INSPIRATION_DURATION_MS := false;
set expiredTIMER_EXPIRATION_DURATION_MS := false;
[...]
step

check time = 1;
set startupEnded := true;
set expiredTIMER_TRIGGERWINDOWDELAY_MS := true;
step

check time = 2;
check state = MAIN_REGION_SELFTEST;
set selfTestPassed := true;
set startupEnded := false;
set expiredTIMER_INSPIRATION_DURATION_MS := true;
step

check time = 3;
check state = MAIN_REGION_VENTILATIONOFF;
set startVentilation := true;
set selfTestPassed := false;
step

check iValve = OPEN;
check state = MAIN_REGION_PCV_R1_INSPIRATION;
check time = 4;
check start(TIMER_INSPIRATION_DURATION_MS) = 3;
set poweroff := true;
set startVentilation := false;
set expiredTIMER_INSPIRATION_DURATION_MS := false;
set expiredTIMER_EXPIRATION_DURATION_MS := true;
step

check iValve = CLOSEDM
check time = 5;
check state = OFF;
set poweroff := false;
set expiredTIMER_INSPIRATION_DURATION_MS := true;
step

[...]
```

Listing 4.8 Executable counterexample in Avalla.

AsmetaSMV can be used to verify the functional correctness of the specified system, and as a back-end tool for other activities supported in ASMETA, e.g., model review.

4.4 ASMETA @ development-time

Once the AsmetaL model is available and verified, the user can automatically generate abstract tests, C++ code, and C++ unit tests. This is an important feature of the ASMETA framework, especially in the context of this book, since the derived code preserves all the properties that have been verified in the ASM model. In this way, if the specification has been correctly evaluated, verified and validated, the generated code is correct-by-construction.

4.4.1 Model-based test generation

Model-based testing [168] is a popular testing approach in which tests are derived, in an automatic manner, from formal models. The technique is based on the consideration that the model is an abstract representation of the System Under Test (SUT), from which it is possible to generate both the test inputs and the expected output (so, tackling the *oracle problem* of software testing [18]). *Abstract tests* are generated starting from the model and then translated into *concrete tests* for the SUT. Coverage criteria over the model are used to define test goals. A typical approach for generating tests that achieve these goals is to use model checkers [84]: a test goal is translated into a suitable temporal property (called *trap property*), whose counterexample (if any) is the test that covers that test goal. More details on how to apply this approach will be given in Chapter 5.

ASMETA integrates the ATGT tool [89] which allows performing model-based test generation using the NuSMV model checker. The generation is guided by coverage criteria defined or adapted for ASMs [88], such as rule coverage, parallel rule coverage, MCDC, etc. For example, the *rule coverage* criterion requires that for every transition rule r_i there exists at least one state in a test in which r_i fires, and another state in a test in which r_i does not fire. Finally, the abstract tests generated with ATGT can be later translated into concrete test cases for the specific implementation. Indeed, the test concretization process may be difficult and needs to be customized for every system under test.

4.4.2 Model-based code generation

According to best practices in model-driven engineering, the implementation of a system should be obtained from its model through a systematic model-to-code transformation, since this allows the proved safety properties to be maintained. Thanks to Asm2C++, given the AsmetaL model of the SUT, the C++ code is automatically generated [38]. This is done by performing a series of steps:

- the AsmetaL specification is parsed and converted into an instance of the ASMETA metamodel (AsmM);

- a *model-to-text* transformation, exploiting Xtext in Eclipse, is applied to translate the model into C++ code.

This procedure generates two files: header (.h) and source (.cpp). The former contains the interface of the source file, the translation of the ASM domains declaration and definition, and functions and rules declaration. The latter contains the rules implementation, the functions/domains initialization, and the definitions of the functions. Listing 4.9 reports an excerpt of the translation in C++ of the MVM specification.

As described in Sect. 4.2.1 an ASM run consists in the execution of the *main rule* and, consequently, in the update of the locations. For this reason, in C++ the ASM step is implemented by two methods:

- `r_Main()`, that corresponds to the translations of the ASM main rule;
- `fireUpdateSet()`, which updates the locations to the next state value.

Given the translation of an `AsmetaL` specification in C++, the code generation process can be easily adapted for a specific platform. ASMETA supports the translation of code in Arduino format (see Sect. 5.3 for a detailed description of how to derive the MVM code for Arduino from the `AsmetaL` model), which is compatible with C++, cheap, and easily accessible. In particular, once the C++ code has been generated, the user is required to perform three additional steps:

- *HW configuration and integration*: the mapping between the ASM functions and the Arduino input / output pins, and other hardware-specific settings must be defined. The first draft of the mapping is automatically generated, but user intervention is required to set all monitored and controlled functions to the correct hardware pins.
- *ASM runner generation*: `Asm2C`++ automatically generates a `.ino` file that contains the `loop()` function to run the translation of the ASM on Arduino. It iteratively executes the following functions: `getInputs()` — reads data from the input devices like sensors; `mainRule()` — contains the behavior described in the `AsmetaL` model; `fireUpdateSet()` — updates the state at the end of each loop; and `setOutputs()` — sets the output values like the current state of light-emitting diode (LED).
- *Merging all the generated files*: all the files previously generated have to be merged and uploaded on the Arduino board.

4.4.3 Unit test generation

If the C++ code is available (either automatically generated or not) and the user wants to test it by using a model-based approach, C++ unit tests can be automatically generated starting from the `AsmetaL` model [37]. Test generation can be performed by exploiting two different mechanisms. The first

```
// MVM0.h automatically generated from ASMETA2CODE
#ifndef MVM0_H
#define MVM0_H

#include <string.h>
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <list>
#include <boost/tuple/tuple.hpp>
using namespace std;
#define ANY String
#define UNDEF NULL

/* DOMAIN DEFINITIONS */
namespace MVM0Namespace{
    enum States {STARTUP, SELFTEST, VENTILATIONOFF,
          PCV_INSPIRATION, PCV_EXPIRATION, OFF};
    enum Modes {PCV};
    enum ValveStatus {OPEN, CLOSED};
}
using namespace MVM0;

class MVM0 : public virtual TimeLibrary{

    /* DOMAIN CONTAINERS */
    const set<States> States_elems;
    const set<Modes> Modes_elems;
    const set<ValveStatus> ValveStatus_elems;

    public:
        /* FUNCTIONS */
        bool poweroff[2];
        bool startupEnded[2];
        bool selfTestPassed[2];
        bool resume[2];
        bool startVentilation[2];
        bool stopVentilation[2];
        Modes mode[2];
        bool pawGTMaxPinsp[2];
        bool dropPAW_ITS_PCV[2];
        bool stopVentilationRequested[2];
        States state[2];
        ValveStatus iValve[2];
        ValveStatus oValve[2];
        [...]

        /* RULE DEFINITION */
        void r_startupEnded();
        void r_ventOffRequested();
        void r_ventOffPCV();
        void r_ventOffFT();
        void r_turnOff();
        void r_PCVInsp();
        void r_PCVInspOValve();
        void r_PCVExp();
        void r_PCVExpIValve();
        void r_Main();

        MVM0();
        void initControlledWithMonitored();
        void getInputs();
        void setOutputs();
        void fireUpdateSet();
};
```

```
// MVM0.cpp automatically generated from ASM2CODE
#include "MVM0.h"

using namespace MVM0Namespace;

// Conversion of ASM rules in C++ methods
void MVM0::r_startupEnded(){
  state[1] = SELFTEST;
}

void MVM0::r_ventOffRequested(){
  stopVentilationRequested[1] = true;
}

void MVM0::r_ventOffPCV(){
    { // par
      state[1] = VENTILATIONOFF;
      stopVentilationRequested[1] = false;
  } // endpar
}

void MVM0::r_ventOffFT() {
    state[1] = VENTILATIONOFF;
}

void MVM0::r_turnOff() {
    { // par
        iValve[1] = CLOSED;
        oValve[1] = OPEN;
        state[1] = OFF;
    } // endpar
}

[...]

void MVM0::r_Main(){
  [...]
}

// Function and domain initialization
MVM0::MVM0(){
    //Static domain initialization
    States_elems:{STARTUP, SELFTEST, VENTILATIONOFF,
          PCV_INSPIRATION, PCV_EXPIRATION, OFF;};
    Modes_elems:{PCV;};
    ValveStatus_elems:{OPEN,CLOSED;};

    // Function initialization
    state[0] = state[1] = STARTUP;
    iValve[0] = iValve[1] = CLOSED;
    oValve[0] = oValve[1] = OPEN;
    [...]
}

// initialize controlled functions that contains monitored functions in
        the init term
void MVM0::initControlledWithMonitored(){
}

// Apply the update set
void MVM0::fireUpdateSet(){
  startupEnded[0] = startupEnded[1];
  selfTestPassed[0] = selfTestPassed[1];
  startVentilation[0] = startVentilation[1];
  [...]
}
```

Listing 4.9: Excerpts of the header and source file automatically generated from the `AsmetaL` model for the MVM case study.

```
BOOST_AUTO_TEST_SUITE(MVM0Test)
BOOST_AUTO_TEST_CASE( my_test_0 ){
        // instance of the SUT
        MVM0 ventilator;
        // state
        // set monitored variables
        ventilator.poweroff[1] = false;
        [...]
        BOOST_CHECK(ventilator.state[0]==STARTUP);
        // call main rule
        ventilator.r_Main();
        ventilator.fireUpdateSet();
        [...]
}
[...]
BOOST_AUTO_TEST_END();
```

Listing 4.10: Example of C++ Boost unit test.

```
#include "catch.hpp"

TEST_CASE( "my_test_0", "my_test_0" ){
        // instance of the SUT
        MVM0 ventilator;
        // state
        // set monitored variables
        ventilator.poweroff[1] = false;
        [...]
        REQUIRE(ventilator.state[0]==STARTUP);
        // call main rule
        ventilator.r_Main();
        ventilator.fireUpdateSet();
        [...]
}
[...]
```

Listing 4.11: Example of C++ Catch2 unit test.

approach consists in running the `AsmetaS` random simulation for a defined number of steps specified by the tester, and in translating the generated state sequence into a C++ unit test. On the contrary, the second approach is based on translating the abstract tests generated by `ATGT` (see Sect. 4.4.1) into C++ unit tests. In both cases, the unit tests can be written using the Boost or Catch2 test frameworks.

An example of the generated unit test is reported in Listing 4.10, if the Boost library is used. Initially, the test suite is registered by using the `BOOST_AUTO_TEST_SUITE(...)` macro. Moreover, the definition of the test suite ends with the macro `BOOST_AUTO_TEST_END()`. Note that each test suite can be composed of one or more test cases, each one declared using the macro `BOOST_AUTO_TEST_-CASE(...)`.

On the other hand, Listing 4.11 shows the same test suite if the Catch2 library is used. Unlike the code with Boost, Catch2 does not require the start and end of the test suite to be specified. Every test case is declared with a `TEST_CASE(testName,tags)` macro.

4.5 ASMETA @ operation-time

In Sect. 4.3.2 formal validation and verification techniques for ASMETA have been presented. These techniques aim at identifying and solving problems at design time. However, the state space of a system under specification is often too large or partially unknown at design time, such as for PEMS with uncertain behavior of humans in the loop (either doctors or patients). This aspect makes complete assurance impractical or even impossible to pursue completely at design time. For this reason, runtime assurance methods can be useful since they take advantage of the fact that variables that are free at design time are, instead, bound at runtime; so, as an alternative to verifying the complete state space, runtime assurance techniques can be focused on checking the current state of a system.

With ASMETA, the developer can use two types of runtime analysis techniques: runtime simulation and runtime monitoring. Both approaches consider the model as a *digital twin* of the real system and exploit it as *oracle* of the correct behavior of the system. In particular, the former exploits the twin execution to prevent misbehavior of the system in case of unsafe model behavior, while the latter exploits the twin execution to check the correctness of the system behavior w.r.t. the model behavior. In this book, ASMETA is never applied at operation-time. However, more details on this use of the framework are available in [7] and an example of its application to PEMS is presented in [39].

4.6 Conclusion

In this chapter, I have presented an overview of the ASMETA model-based analysis approach and the associated tooling for the safety assurance problem of PEMS, using ASMs as the underlying analysis formalism and complying with all activities required by the certification standards (see Sect. 1.2). ASMETA is an active open-source academic project. Over the years, it has been improved with new techniques and tools to face the upcoming new challenging aspects of modern systems. In accordance with the topic of this book, in the next chapter, I will present some application scenarios in which ASMETA and its tools have been used for the safety assurance of medical systems, such as for a pill-box, the MVM (previously described in Chapter 2) and the PHD protocol.

Chapter 5. Applying the ASMETA rigorous process to medical case studies

In this chapter, I present the application of MBSE techniques to real medical systems and software. In particular, the ASMETA framework (previously introduced in Chapter 4) is applied for test and code generation, validation, and verification in three different case studies: the e-Pix medicine reminder, the MVM (see Chapter 2) and the PHD medical communication protocol. The activities presented aim to be compliant with those required by the IEC 62304 [64] standard and FDA guidelines [83] for software validation, as presented in Sect. 1.2 of this book.

This chapter is based on the work published in [25–29] and is structured as follows. Sect. 5.1 introduces the activities that can be carried out on medical devices by exploiting the ASMETA framework, while in Sect. 5.2 the case studies on which these activities are performed are introduced, namely the e-Pix pill box, the MVM, and the PHD protocol. Then, Sect. 5.3 tackles the problem of generating code from the ASMETA specification of the systems analyzed, while Sect. 5.4 presents the approach that can be used to derive tests to be executed on the actual medical system from the ASMETA specification. Finally, Sect. 5.5 concludes the chapter.

5.1 Introduction

The development of medical software and systems must adhere to certification standards, in order to ensure the safety and reliability of each device interacting with human beings. In Sect. 1.2, I have presented the two main standards for medical software, namely IEC 62304 [64] and the FDA guidelines [83]. In general, both documents require the software to be tested and verified at each development step, and developed through a well-documented software life cycle. Furthermore, in Chapter 4, I have presented the ASMETA framework and the activities that it supports during the development of safety-critical systems.

The idea underlying this chapter is that by using the ASMETA framework and its tools, one can fulfill the majority of the activities required by certification standards.

In the following, I map the two main documents (IEC 62304 and FDA guidelines) to the ASM-based development process using the ASMETA framework. Tab. 11 reports the mapping of ASMETA activities to those required by Section 5 of the IEC 62305 [64] standard, which is focused on the characteristics of the software development process.

Step	Activity description	ASMETA mapping
5.1	Define a life cycle model and plan all procedures.	ASMs provide a precise, iterative, and incremental life-cycle model, based on model refinement. With ASMs, developers can perform modeling, validation, verification, and conformance checking.
5.2	Define and document functional and non-functional software requirements.	System requirements can be defined using the ASM notation, a mathematical model that can also be analyzed and checked before the implementation. However, only functional requirements can be natively modeled with ASM, so non-functional ones should be analyzed with complementary techniques.
5.3	Specification of the software architecture from the software requirements, risk-control activities and verification.	Verification of software requirements can be carried out throughout the ASM development process using the property verification tool `AsmetaSMV`. Risk control activities can be carried out by verifying the required functional safety properties and performing critical scenario-based tests written in `Avalla`.
5.4	Refine the software architecture into software units.	The software refinement can be obtained by means of the model refinement mechanism, typical of the ASMETA approach. The correctness of refinement can be proved by using the `ASMRefProver` tool.
5.5-5.7	Software implementation and testing at the unit, integration, and system levels.	With the ASMETA development process, the actual code can be obtained using the automatic translator `Asm2C++` on the last step of model refinement. Since verification and validation activities should be carried out at model-level, the code obtained with the automatic translator is correct by construction. However, the developer may change something in the generated code (since some of the aspects may not be representable with the ASM formalism), so the ASM process cannot fully cover these development steps. Even when the implementation is already available and the model has been written only for verification and validation purposes, tests can be derived from the ASM and executed on the real system.
5.8	Demonstration, by the device manufacturer, that the software has been validated and verified.	If the ASM process is used, the demonstration that the software has been validated and verified is straightforward, since V&V activities are continuous activities during the entire process. Moreover, these activities can be reexecuted at any time since they are automatically performed on the ASM models.

Table 11: Mapping between IEC 62304 and ASMETA activities.

The presented mapping shows how all points are covered, at least partially, by activities composing the ASM-based development lifecycle.

Guideline description	**ASMETA mapping**
A documented software requirements specification should provide a baseline for both V&V.	In ASM, the software requirements specification is written using a formal model (or a chain of models, for complex systems), that can be used for performing V&V activities by using the ASMETA tool set.
Developers should use a mixture of methods and techniques to prevent and detect software errors.	In ASM, safety properties can be proved at every refinement level, using the `AsmetaSMV` tool. If developers specify correctly the properties, errors can be easily detected when a property cannot be demonstrated (or it has been demonstrated to be false). Furthermore, specification errors can be revealed even using scenario-based testing.
Software V&V should be planned and conducted during all the software life cycle.	The ASMETA V&V process can be applied to each model, and the activities can be integrated either in the V model or in the agile software development life cycles. In particular, it is possible to insert V&V activities in the modules design, coding, and unit testing phases, both at the model and code level.
Software V&V processes should be executed through the use of procedures.	In ASM, V&V activities are supported by precise procedures defined for each tool of the ASMETA framework. These activities can be re-executed when needed, since they are automatically performed.
Software V&V should be re-established upon any software change	In ASM, if software changes do not affect the model, testers must rerun unit tests (which can be generated from the ASM model) on the changed software and verify if the behavior is still correct. On the contrary, in case the software changes have effects on the model, V&V activities can be executed at the model level.

Validation coverage should be based on software complexity and safety risks	Tools embedded into the ASMETA framework allow producing a report on coverage (in terms of rules) during validation. This information can be used by the designer to estimate whether the validation activity is commensurate with the risk associated with the use of the software.
V&V activities should be carried out using the quality assurance precept of "independence of review"	This aspect is implicit in ASMETA since V&V activities are executed with unambiguous mathematical-based techniques.
The device manufacturer has flexibility in choosing how to apply the V&V principles contained into the FDA guidelines	All the ASMETA V&V activities can be executed at the discretion of the manufacturer, because they can be executed independently of each other.

Table 12: Mapping between FDA guidelines and ASMETA activities.

On the other hand, the FDA guidelines [83] accept the standard IEC 62304 and push even further for the integration of software life cycle management and risk management activities. Tab. 12 reports the mapping of ASMETA activities to those required by the FDA guidelines. As well as for IEC 62304, the FDA guidelines can be easily mapped to the ASMETA principles and activities.
After having described the mapping between certification standards and the ASMETA framework, in this chapter, ASMETA and its tools are exploited for modeling, validation, verification, and test and code generation for medical devices.

5.2 Case studies

MBSE activities have been applied to a variety of systems, both in the medical domain and in other domains. However, in this book, I present the activities carried out for three different case studies in the domains of PEMS, namely, MVM, e-Pix and PHD communication protocol. The first case study has been extensively discussed and presented in Chapter 2. Thus, in this section, I present the two remaining case studies by giving more details about their functioning and scope.

5.2.1 The e-Pix case study

Adherence to pharmacological therapy [51] is one of the most well-known problems in the medical field. In fact, sometimes, patients do not adhere to therapy because they do not remember to take the medicine, or do not remember if they have already taken it. For these reasons, some patients may have the need to adopt a system that can help them in following the prescribed therapy. Recently, pill boxes have been proposed on the market. They contain pills and divide them according to scheduled doses of medications. The aim of a pill box is to help the user to prevent/reduce medication errors because once the pills are in the correct section, the user only has to remember to take them at the right time. The first versions were simply multi-compartmental boxes, where each compartment was filled with the corresponding medicine. They may have one section for each day, or, in the case of the most complicated versions, multiple sections corresponding to different times of the day. With the introduction of technology in the medical field, even pill boxes have evolved and are integrated with electronic components in order to provide alerts to patients when the time of a medicine comes. They are usually provided with a memory, where the list of pills with the therapy schedules is saved and, at the right time, they notify the user. Different types of notifications can be used, e.g., sound/light signals, or pop-up notifications on a smartphone.

The activities discussed in this chapter have been carried out on a pill box, called e-Pix, developed using Arduino[1] that a local company asked to re-engineer. The need for reengineering comes from the fact that the company wanted to certify its product w.r.t. the FDA guidelines [83] and IEC regulation [64] and, because of that, needed to be sure it works properly. In the following, some requirements (even not directly related to software certification) and properties of the e-Pix device are reported:

- Each compartment of e-Pix contains a unique type of unpackaged pills;
- Each compartment has a sensor capable of signaling the opening of the related drawer, and a red LED used to indicate which pill has to be taken. The LED turns on until the patient opens the compartment;
- When the pill time has passed, and the set timeout has expired, the red LED starts to blink for a defined period of time. In this way, e-Pix attracts the patient's attention;
- e-Pix has an embedded display that shows log messages;
- If a patient takes the pill but forgets to close the drawer, the red LED starts to blink for a fixed period of time;
- The prescription file can be loaded into e-Pix by transferring it using the integrated Bluetooth communication functionalities. The file contains, for each pill, the identifier of the drawer in which it is contained, its name, and the time at which it has to be taken (expressed as the number of seconds passed since 01/01/1970). An example of this file is reported in Listing 5.1.

[1]https://www.arduino.cc/

```
{
    “patient” : “patient_name”,
    “pills” : [
                    { “compartment” : “compartment_number”,
                            “name” : “pillName”,
                            “time_consumption” : [“t1”, “t2”, ...],
                    },
                    {...}
        ]
}
```

Listing 5.1: Example of the JSON file containing the prescriptions.

5.2.2 The IEEE 11073 PHD protocol case study

The IEEE 11073-20601 [102] standard defines a communication protocol that allows personal healthcare devices defined as “Agents”, which are normally portable, energy-efficient, and have limited computing capacity (such as weighing scales, blood glucose monitors, and blood pressure monitors), to exchange information with devices with more computing resources, defined as “Managers” (such as mobile phones, set-top boxes, and personal computers). The information exchanged is basically measured health data that can be transmitted to healthcare professionals for different purposes, e.g., for remote health monitoring or health advising.

The PHD IEEE 11073 standard defines both the data exchange protocol and the necessary data models to be used during the communication between two devices. The messages exchanged are called APDUs, are encoded using the ASN.1 format, and should support at least the MDER (Medical Device Encoding Rules) standard. IEEE 11073 requires communication to have one reliable primary virtual channel and some additional secondary virtual channels. The message types are divided into four different categories:

- messages used during the association procedure: aare (Association Request), aarq (Association Response), rlre (Association Release Response), rlrq (Association Release Request), abrt (Association Abort);
- messages related to the confirmed service mechanism: roiv-* (Remote Operation Invoke messages): roiv-cmip-confirmed-action, roiv-cmip-confirmed-event-report, roiv-cmip-confirmed-set; and rors-* (Reception of Response messages): rors-cmip-confirmed-action, rors-cmip-confirmed-event-report, rors-cmip-get;
- messages used when fault or abnormal conditions occur: roer (Reception of Error Result), rorj (Reception of Reject Result);
- messages related to the unconfirmed service mechanism: roiv-cmip-action, roiv-cmip-event-report, roiv-cmip-set.

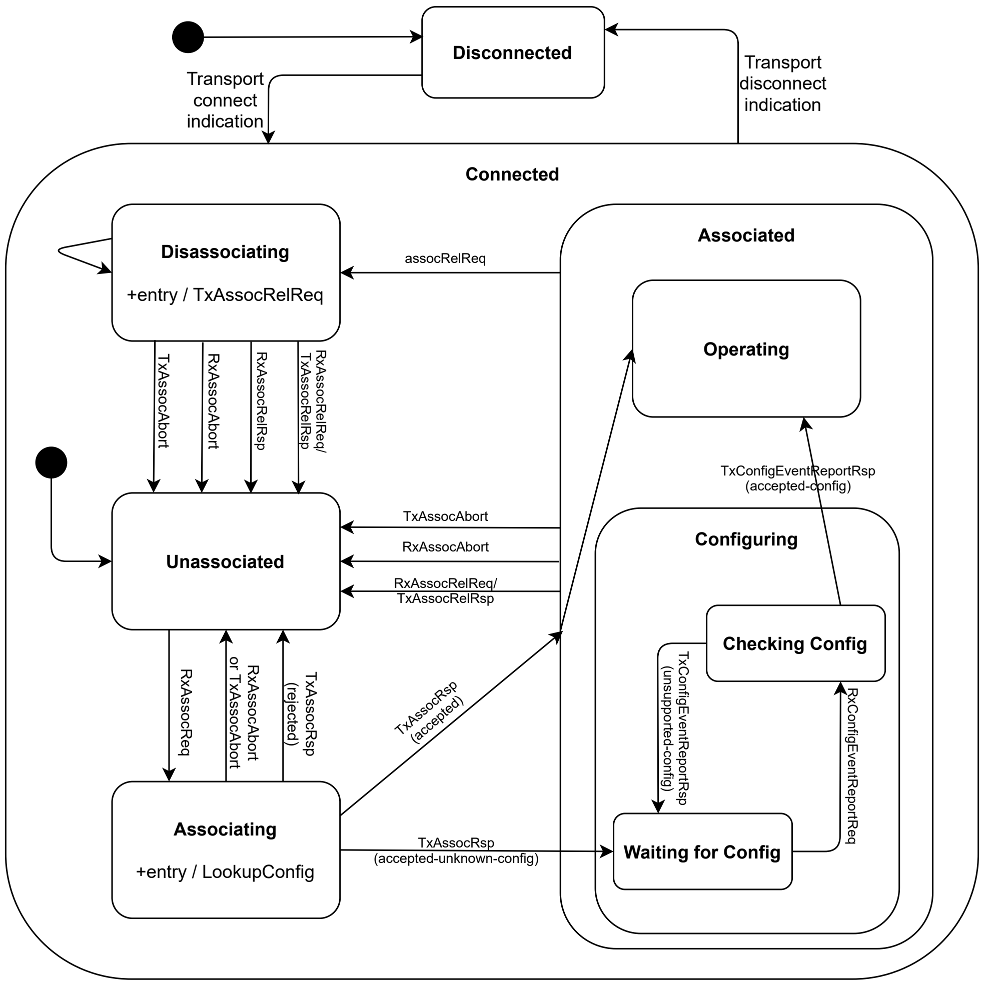


Figure 23: State machine of the IEEE 11073 PHD Manager: input messages are identified by the prefix Rx and output messages are identified by the prefix Tx (when no input message is associated to an output, it means that the transition is generated by an internal event).

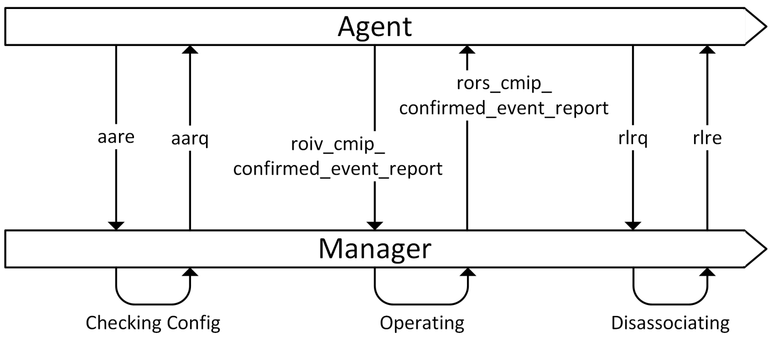

Agent
aare
aarq
roiv_cmip_ confirmed_event_report
rors_cmip_ confirmed_event_report
rlrq
rlre
Manager
Checking Config
Operating
Disassociating

Figure 24: An example sequence of data exchange using the PHD protocol.

The behavior of the PHD protocol, in the case of a manager device, is described by the state machine in Fig. 23, which is composed of seven states. To better understand the behavior of the protocol and how a device moves from one state to another, Fig. 24 reports an example scenario of a weighting scale. The weighting scale (which acts as *agent*) sends an association request message to the manager containing device configuration information. Then, the manager checks the received information: if the manager recognizes the agent configuration (*checking config* internal state), it sends a response certifying the association acceptance, and both devices enter the *Operating* state. Suppose now that the weighting scale is ready to communicate the measured information: the agent sends the measured data to the manager using a *Confirmed Event Report* APDU, and, if the message is received correctly by the manager, the manager responds with the acknowledgment. Finally, when the agent has communicated all the measures, it requests to release the association; the manager responds to this request and both devices now enter the *Unassociated* state.

5.3 From ASMETA specifications to embedded code

Starting from the ASMETA specifications, `Asm2C++` allows users to obtain a C++ code that can be embedded in the actual device or in Arduino. The main advantage of this approach is that ASMETA specifications can be validated and verified and, therefore, the C++ code obtained is correct-by-construction. This approach is based on the following iterative process:

1. Model the system as an Abstract State Machine;
2. Validate and verify the model;
3. If the model has proved to be correct, then start refining it by adding more details;
4. Prove that the refinement has been done correctly, such that all the verified properties are still verified;
5. Repeat the two previous steps until all the relevant aspects have been modeled;
6. When the final model is available, convert it into C++ code.

In the following, the depicted process is applied to two different case studies, namely the e-Pix and the MVM, from the ASMETA specification to the Arduino code.

5.3.1 The e-Pix case study

In this section, the ASMETA process, from specification to code, is applied to the e-Pix case study (previously presented in Sect. 5.2.1). For each phase, here I report only some example. The full specifications, scenarios, and code are available online at `https://foselab.unibg.it/asmeta/PillboxASM.zip`.

Modeling by refinement

As commonly done in the ASM-based development process, the e-Pix has been modeled starting from a simple model and then applying step-wise refinement. From one refinement step to the next, some controlled and monitored functions have been added, mainly representing compartments and time management mechanisms: at level 0 the time is represented in an abstract way, it is managed through a controlled variable at levels 1 and 2, and with a monitored function at the final level.

In the following, I report the main characteristics of each refinement level and, as an example, analyze how the switching ON of the red LED when the time of a pill comes has been modeled.

- **Level 0**: this level models only a single pill, with a single prescription, and no compartments. The time is managed by a simple Boolean monitored function *takeThePill*, which becomes true when the pill has to be taken. A similar approach has been used to represent all timeouts (e.g., to manage the change in the color of the LED), by using another Boolean function *timeDiffOver600* (we consider the timeouts being 10 minutes). The following `AsmetaL` rules manage the behavior of the red LED that is turned on when a pill needs to be taken:

```
main rule r_Main =
  [...]

  if redLed = OFF and takeThePill then
    r_pillToBeTaken[] endif
  if redLed = ON and not timeDiffOver600 and
  opened and not openSwitch then
    r_pillTaken_compartmentOpened[] endif

  [...]
```

```
rule r_pillToBeTaken =
  par
    redLed := ON
    outMess := TAKE_PILL
  endpar
rule r_pillTaken_compartmentOpened =
  par
    redLed := OFF
    outMess := NONE
  endpar
```

- **Level 1**: the main addition at this level has been an improvement in time management. In fact, the Boolean function has been substituted with the Natural function *systemTime*, which is controlled by the system and increased at each machine step. Other aspects, such as the number of pills, prescriptions, or compartments, remain unchanged w.r.t. the previous level. To manage the assumption of the pill, a new Boolean function, *requestSatisfied*, identifies if the pill has already been taken or not. This refinement level also adds possible output and log messages, which are taken from an enumerative domain *OutMessages*. The following `AsmetaL` rules manage the behavior of the red LED that is turned on when a pill has to be taken, depending on *systemTime*:

```
main rule r_Main =
  [...]
  if redLed = OFF and (time_consumption<=systemTime
  and not requestSatisfied) then
      r_pillToBeTaken[]
  endif
  [...]
```

```
rule r_pillToBeTaken =
  par
      if redLed != ON then
      compartmentTimer := systemTime endif
      redLed := ON
      outMess := TAKE_PILL
  endpar
```

- **Level 2**: the second refinement level introduces three compartments, each with a single type of pill. Other features are similar to the previous level: a single deadline is used for each pill, the output and log messages come from the same enumerative domain *OutMessages*, the timer *systemTime* is managed by the system and takes values in a bounded range. Since many compartments need now to be controlled, the rules managing the behavior of the red LED, that is switched on when a pill has to be taken, now change as follows:

```
main rule r_Main =
[...]
if redLed($compartment) = OFF and
  (time_consumption($compartment)<=systemTime and not requestSatisfied($compartment)) then
  r_pillToBeTaken[$compartment]
endif
[...]

rule r_pillToBeTaken($compartment in Compartment) =
par
  if redLed($compartment) != ON then compartmentTimer($compartment) := systemTime endif
  redLed($compartment) := ON
  if ($compartment=compartment1) then
    outMess($compartment) := TAKE_TYLENOL
  else if ($compartment=compartment2) then
    outMess($compartment) := TAKE_ASPIRINE
  else
    outMess($compartment) := TAKE_MOMENT
  endif endif
endpar
```

- **Level 3**: this refinement level includes all the features specified by system requirements:
  - all three compartments have been modeled;
  - the *systemTime* is monitored from the machine and updated by the environment (its management can be configured by users as presented in Sect. 4.3.1);
  - log and out messages can be any string;

  – a list of time prescription (stored in the function *time_consumption*) can be assigned to each compartment.

The guard that makes the red LED turn on when it is time to take the pill has been modified w.r.t. the previous levels because the model now manages more prescriptions for each pill. Considering *time_consumption* the list of prescriptions, the correct item in the sequence, i.e., the current time threshold to be considered, is selected with the function *drugIndex*. Therefore, for the compartment *d*, when *systemTime* passes the function *time_consumption* in position *drugIndex(d)*, the pill in *d* should be taken.

```
main rule r_Main =
[...]
  if redLed($compartment) = OFF and
  (at(time_consumption($compartment),drugIndex($compartment))<systemTime) then
    r_pillToBeTaken[$compartment]
endif
[...]

rule r_pillToBeTaken($compartment in Compartment) =
par
  if redLed($compartment) != ON then
    compartmentTimer($compartment) := systemTime endif
  redLed($compartment) := ON
  outMess($compartment) := "Take␣" + name($compartment)
endpar
```

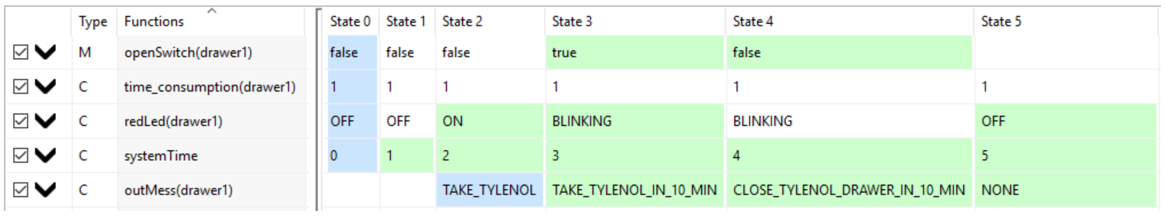
Refinement proof

`AsmRefProver` can prove the correctness of the model refinement process by exploiting the Satisfiability Modulo Theories (SMT). It takes as input two different models, representing two refinement levels of the same system, and allows for ensuring that an ASM specification is a correct refinement of a more abstract one.

Since `AsmRefProver` maps refined functions to abstract ones with the same name, in the e-Pix case study, it has been necessary to introduce some derived functions that represent predicates over the abstract or refined states. For example, in the first refinement step, to allow `AsmRefProver` to prove the correctness of the refinement, two derived functions have been added:

- *takeThePill*: indicates if the patient has to take a pill;
- *timeDiffOver600*: becomes true when the patient has forgotten to take the pill within a certain time.

Formally, they are defined as follows:



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5
M	openSwitch(drawer1)	false	false	false	true	false	
C	time_consumption(drawer1)	1	1	1	1	1	1
C	redLed(drawer1)	OFF	OFF	ON	BLINKING	BLINKING	OFF
C	systemTime	0	1	2	3	4	5
C	outMess(drawer1)			TAKE_TYLENOL	TAKE_TYLENOL_IN_10_MIN	CLOSE_TYLENOL_DRAWER_IN_10_MIN	NONE

Figure 25: Simulation steps with the animator `AsmetaA` at the last refinement level for the e-Pix case study.

```
function takeThePill = (time_consumption<=systemTime)
function timeDiffOver600 = (systemTime−compartmentTimer>tenMinutes)
```

Validation

During validation activities, the user can analyze the ASMETA specification using the simulator `AsmetaS`, the animator `AsmetaA`, and the model advisor `AsmetaMA`. The animator is the tool that is generally more used, since it provides a graphical interface which is more readable to the user during model execution.

Fig. 25 reports some of the simulation steps using the animator `AsmetaA`: after the system initialization, when the time is controlled by the ASM and only one pill in the first compartment is available, the red LED turns ON when it is time to take the pill ($systemTime > time\_consumption$) and turns to BLINKING when the timeout has passed; then, when the compartment is closed, the red LED turns OFF; in the meantime, the message $outMess$ shown on the display changes accordingly.

Scenario-based testing

In the scenario-based testing activity, the behavior of e-Pix has been checked against the expected one by simulating all possible states and transitions between them. As introduced in Sect. 4.3.2, the scenarios are written in the `Avalla` language and executed through the validator `AsmetaV`. It checks if, at each step, the machine runs as expected and allows one to evaluate the coverage of each scenario. In particular, for this case study, it was verified that the coverage obtained in terms of the rules of the ASM models was 100%. This is an important result, since it allows users to increase the confidence in the correctness of the written model.

An excerpt of a scenario for the e-Pix case study is shown in Listing 5.2. Initially, all the compartments are closed and, after an ASM step, the red LED is off, and no messages are shown. When the time to take the pill is reached (“step until” command), the state changes, the red LED turns on, and the message shows which pill the patient has to take.

```
// Setting-up the initial state
set openSwitch(comp1) := false;
set openSwitch(comp2) := false;
set openSwitch(comp3) := false;

step

check redLed(comp1) = OFF;
check outMess(comp1) = NONE;
check logMess(comp1) = NONE;

// Time to take the pill in comp1
step until systemTime = 2;

check redLed(comp1) = ON;
check outMess(comp1) = TAKE_TYLENOL;
check logMess(comp1) = NONE;
```

Listing 5.2: Example of `Avalla` scenario for the e-Pix case study.

Property verification

Once the modeler is sufficiently confident that the model correctly reflects the intended requirements, heavier techniques can be used for property verification. In this case study, the following four CTL (Computational Tree Logic) properties have been identified:

1. If the pill has to be taken, the red LED must light up;
2. If the patient does not take the pill or the compartment has to be closed, the red light has to blink;
3. The red light has to change status after 10 minutes if the patient does not take the pill;
4. If the patient takes the pill and closes the compartment, the red light turns off.

After having defined the CTL properties, the `AsmetaSMV` tool translates the ASM specifications into SMV models, which are verified using the NuSMV[2] model checker.
Note that the same property may have a different way of being expressed from a refinement level to the next one. For example, Tab. 13 reports the first property for all models, while the others are available online in the replication package.
In this case, the presented property is different from one model to the other because the time has been managed differently: initially, it has been modeled with a monitored function, then with the function systemTime controlled by the system and increased at each machine step. Furthermore, another difference is due to the fact that in the last refinement step shown (step 2), more compartments have been modeled and, for this reason, the property has been verified over each compartment.

[2]`http://nusmv.fbk.eu/`

Ref. step	CTL Property
0	**ag**((takeThePill and redLed = OFF) **implies** **ax**(redLed = ON))
1	**ag**((takeThePill and not requestSatisfied and redLed = OFF) **implies** **ax**(redLed = ON))
2	(**forall** $d **in** Compartment with **ag**((time_consumption($d)¡systemTime and not request-Satisfied ($d) and opened($d) and not(openSwitch($d)) and not(redLed($d)= OFF) and not(systemTime-compartmentTimer($d)¿=tenMinutes)) **implies** **ax**(redLed($d)= OFF)))

Table 13: The first property in different refinement levels.

Unfortunately, it is not possible to test the property on Level 3 because the model contains unlimited domains (such as natural numbers and strings) that are not supported by the NuSMV model checker. In this case, other model checkers, such as NuXMV should be used. However, the support of ASMETA for NuXMV is still under development.

C++ code generation

After having modeled, verified, and validated the e-Pix ASMETA specification, its C++ code can be generated by exploiting the `Asm2C++` tool. In fact, e-Pix is a prototype of an Arduino-based medical device and is composed of the following components:

- Arduino Mega 2560;
- 3 reed switches, used to signal the opening of each compartment;
- 3 red LEDs to indicate the state of each compartment;
- 1 LCD to show output messages;
- 1 DS3231 timer module to get the current time;
- Arduino Bluetooth module, used to allow the communication with external devices and to receive the JSON file containing the drug prescriptions from the patient's smartphone;
- Arduino SD card reader, used to read and write on the SD card storing the JSON prescription and log files;
- Several potentiometers and resistors;

`Asm2C++` generates, from the last ASM refinement level, the following files: I) the `ino`, which contains the execution policy to run an ASM on Arduino (see Listing 5.3); II) the `a2c` and the `hw.cpp` files that contain hardware information; III) the `h` and `cpp` files, which contain the translation of the ASM model into C++ code.

The `a2c` configuration file is used to link each ASM function to the physical pins of Arduino and is automatically generated. However, user intervention is needed since it is necessary to fill in the mappings with those corresponding to the hardware configuration (see Listing 5.4). Finally, `Asm2C++`

```
#include"pillbox.h"
void setup(){
}

pillbox ePix;

void loop(){
  ePix.getInputs();
  ePix.r_Main();
  ePix.fireUpdateSet();
  ePix.setOutputs();
}
```

Listing 5.3: Example of the ino file containing the implementation of ASM execution for e-Pix.

```
{
  "arduinoVersion": "MEGA2560",
  "stepTime": 0,
  "bindings": [
    {
      "mode": "DIGITAL",
      "function": "redLed(comp1)",
      "pin" : "D1"
    },
    {
      "mode": "DIGITAL",
      "function": "redLed(comp2)",
      "pin" : "D2"
    },
    [...]

  ]
}
```

Listing 5.4: Example of the `a2c` configuration file.

```
#include "pillbox.h"
#include <Arduino.h>
void pillbox::getInputs(){
        openSwitch[comp1] = (digitalRead(7) == HIGH);
        [...]
        systemTime = analogRead(A1)*(double)(1.0/1024.0);
}
void pillbox::setOutputs(){
        if(redLed[1][comp1] == OFF)
                digitalWrite(1, LOW);
        else
                digitalWrite(1, HIGH);
        if(redLed[2][comp1] == OFF)
                digitalWrite(2, LOW);
        else
                digitalWrite(2, HIGH);
        [...]
}
```

Listing 5.5: Example of the `hw.cpp` file.

generates the `hw.cpp` file, which contains the C++ code used to read the inputs and set the outputs (see Listing 5.5).

5.3.2 The MVM case study

As previously done for the e-Pix case study, in the following, the ASM-based development process is applied to the MVM case study (see Chapter 2 for further details on the device), starting from the ASM specification to the C++ code. In this case, even if the real device was already available, the experiments have been carried out on a prototype, based on Arduino, that has been developed in order to conduct experiments on mechanical ventilators. Note that the presented activity has been performed only on a sub-component of the MVM, namely the *controller*, which is the most important part for the device functioning since it manages the passage between ventilation states.

```
main rule r_Main =
    par
        if state = STARTUP then r_startup[] endif
        if state = SELFTEST then r_selftest[] endif
        if state = VENTILATIONOFF then r_ventilationoff[] endif
        if state = PCV STATE then r_runPCV[] endif
        if state = PSV STATE then r_runPSV[] endif
    endpar
```

Listing 5.6: Main rule for the first refinement level of the MVM.

Modeling by refinement

As done for the e-Pix, also the MVM has been modeled starting from a simple model capturing basic behaviors and then applying step-wise refinement. From one refinement level to the other, new functionalities have been added.

In the following, I report the main characteristics of each refinement level and some excerpts from the ASMETA specification representing the newly added functionalities.

- **Level 0**: this first level introduces the main phases of the MVM functioning. In this model, at the end of the *startup* and *self-test* phases, the ventilator moves into the *ventilation off* state. Then, upon user's request, the device can start ventilating either in PCV or PSV. The main rule of this model is reported in Listing 5.6. The controller specifies the transitions among the ventilator states by assigning the corresponding value to the *state* variable. The rule to be executed is then chosen, depending on the state.

- **Level 1**: this refinement level adds details to the inspiration and expiration phases in both the PCV and PSV modes. Listings 5.7 and 5.8 show, respectively, the refinement of the rules used to perform PCV and PSV ventilation. In this case, both the rules have been split into two different sub-rules: one for the inspiration and one for the expiration. During PCV, the transition between inspiration and expiration is controlled by the duration of each phase decided by the physician (when the timers *timerInspirationDurPCV*, in the case of inspiration, and *timerExpirationDurPCV*, in the case of expiration, expire). When *timerInspirationDurPCV* runs out (Listings 5.7 line 12), the controller moves to the PCV expiration phase (line 15). However, if the physician has set *respirationMode* to PSV, the MVM starts the expiration in PSV mode by executing the rule `r_PSVStartExp` (line 17). If the stop of ventilation is requested (function *stopRequested*) during the inspiration phase, the function *stopVentilation* is set, and the stop is actually performed only during the expiration phase (line 9). When the MVM is in PCV expiration, the ventilator moves to PCV inspiration after *timerExpirationDurPCV* expires.

```
rule r_runPCV =
  par
    if phase = INSPIRATION then r_runPCVInsp[] endif
    if phase = EXPIRATION then r_runPCVExp[] endif
  endpar

rule r_runPCVInsp =
  par
    if not stopVentilation then
      if stopRequested then stopVentilation := true endif
    endif
    if expired(timerInspirationDurPCV) then
      par
        if respirationMode = PCV then
          r_PCVStartExp[]
        endif
        if respirationMode = PSV then
          par
            state := PSV_STATE
            r_PSVStartExp[]
          endpar
        endif
      endpar
    endif
  endpar

rule r_runPCVExp =
  if stopVentilation then r_stopVent[]
  else if stopRequested then r_stopVent[]
  else if expired(timerExpirationDurPCV) then
    r_PCVStartInsp[]
  endif endif  endif

rule r_PCVStartInsp =
  par
    phase := EXPIRATION
    iValve := CLOSED
    oValve := OPEN
    r_reset_timer[timerInspirationDurPCV]
  endpar
```

Listing 5.7: PCV management for the second level of refinement of the MVM.

```
rule r_runPSV =
  par
    if phase = INSPIRATION then r_runPSVInsp[] endif
    if phase = EXPIRATION then r_runPSVExp[] endif
  endpar

rule r_runPSVInsp =
  par
    if not stopVentilation then
      if stopRequested then stopVentilation := true endif
    endif
    if (expired(timerMinInspTimePSV) and flowDropPSV)
      or expired(timerMaxInspTimePSV) then
      r_PSVStartExp[]
    endif
  endpar

rule r_runPSVExp =
  if stopVentilation then r_stopVent[]
  else if stopRequested then r_stopVent[]
  else if expired(timerMinExpTimePSV) then
    par
      if respirationMode = PCV then
        par
          state := PCV_STATE
          r_PCVStartInsp[]
        endpar
      endif
      if respirationMode = PSV then
          r_PSVStartInsp[] endif
    endpar endif endif endif

rule r_PSVStartInsp =
  par
    phase := EXPIRATION
    iValve := CLOSED
    oValve := OPEN
    r_reset_timer[timerMinExpTimePSV]
    r_reset_timer[timerMaxInspTimePSV]
  endpar
```

Listing 5.8: PSV management for the second level of refinement of the MVM.

On the contrary, during PSV, the transition from inspiration to expiration occurs when the airflow drops below a defined threshold *flowDropPSV* (Listings 5.8 at line 13) and the minimum inspiration time has passed or when the maximum inspiration time set by the doctor runs out. Instead, the transition from expiration to inspiration is performed after the expiration of *timerMinExpTimePSV* (line 21). As for the PCV mode, the stop of ventilation can be performed only during expiration. In addition, the physician can change the mode from PSV to PCV without interrupting ventilation, when in the expiration phase (line 23), if the patient's condition requires additional support.

- **Level 2**: this refinement level introduces the inspiratory pause, the expiratory pause, and the recruitment maneuver, which can be manually requested by the physician. Listings 5.9 and 5.10 show how these three functions have been modeled, respectively, for the PCV and PSV mode.

```
rule r_runPCV =
  par  [...]
    if phase = INPAUSE then r_runInPause[] endif
    if phase = RM then r_runRm[] endif
    if phase = EXPAUSE then r_runExPause[] endif
  endpar

rule r_runPCVInsp = [...]
  if expired(timerInspirationDurPCV) then
  par
      if respirationMode = PCV then
        if cmdInPause then
          r_InPause[]
        else
          if cmdRm then  r_rm[]
          else r_PCVStartExp[]
        endif
      endif endif
      if respirationMode = PSV then
        par
          state := PSV_STATE
          r_PSVStartExp[]
          r_resetApneaBackup[]
        endpar
      endif
    endpar
  endif [...]

rule r_runPCVExp =  [...]
  if expired(timerExpirationDurPCV) then
    if cmdExPause then
      r_exPause[]
    else
      r_PCVStartInsp[]
    endif
  endif
  [...]
```

Listing 5.9: PCV management for the third level of refinement of the MVM.

```
rule r_runPSV =
  par  [...]
    if phase = INPAUSE then r_runInPause[] endif
    if phase = RM then r_runRm[] endif
    if phase = EXPAUSE then r_runExPause[] endif
  endpar

rule r_runPSVInsp =  [...]
  if (expired(timerMinInspTimePSV) and flowDropPSV)
    or expired(timerMaxInspTimePSV) then
  if cmdInPause then r_InPause[]
  else if cmdRm then r_rm[]
  else r_PSVStartExp[] endif endif
  endif [...]

rule r_runPSVExp = [...]
  if expired(timerApneaLag) then r_runApnea[]
  else  if expired(timerMinExpTimePSV) then
    par
      if respirationMode = PCV then
        par
          state := PCV_STATE
          r_PCVStartInsp[]
        endpar
      endif
      if respirationMode = PSV then
        if cmdExPause then r_ExPause[]  endif
      endif
    endpar
  endif endif [...]

rule r_runApnea =
  par
    state := PCV_STATE
    r_PCVStartInsp[]
    apneaBackupMode := true
  endpar
```

Listing 5.10: PSV management for the third level of refinement of the MVM.

The inspiratory pause is required when *cmdInPause* is set (see Listing 5.9 at line 12 and Listing 5.10 at line 11). In this case, both valves are closed for the entire duration of the pause. The recruitment maneuver is required when the *cmdRm* is set (see Listing 5.9 at line 15 and Listing 5.10 at line 12). When this functionality is enabled, the lungs are filled with oxygen and medical air, the output valve is closed, and the input valve is opened to allow air to flow into the alveoli. Finally, the inspiratory pause is required when the *cmdExPause* is set (see Listing 5.9 at line 31 and Listing 5.10 at line 27). Furthermore, when the MVM ventilates in PSV mode, if a new breath is not detected within the expiry of *timerApneaLag* (Listing 5.10 at line 35), the ventilator automatically switches to PCV mode from the inspiration phase. This behavior is needed to avoid apneas.

- **Level 3**: this last refinement level introduces the transition from inspiration to expiration and vice versa, when the pressure changes due to spontaneous breathing. The new behavior has been

```
rule r_runPCVInsp =
    [...]
  if expired(timerInspirationDurPCV) then
    [...]
  else if pawGTMaxPinsp then
    r_PCVStartExp[]
  endif endif

rule r_runPCVExp =
    [...]
  if expired(timerExpirationDurPCV) then
    [...]
  else if expired(timerTriggerWindowDelay)
  and dropPAW_ITS then
    r_PCVStartInsp[]
  endif endif [...]
```

Listing 5.11: PCV management for the fourth refinement level of the MVM.

```
rule r_runPSVInsp =
    [...]
  if (expired(timerMinInspTimePSV) and flowDropPSV) or
   expired(timerMaxInspTimePSV) then
    [...]
  else if pawGTMaxPinsp then
    r_PCVStartExp[]
  endif endif

rule r_runPSVExp =
    [...]
  if expired(timerTriggerWindowDelay)
  and dropPAW_ITS then
    r_PSVStartInsp[]
  else if expired(timerApneaLag) then
    [...]
```

Listing 5.12: PSV management for the fourth refinement level of the MVM.

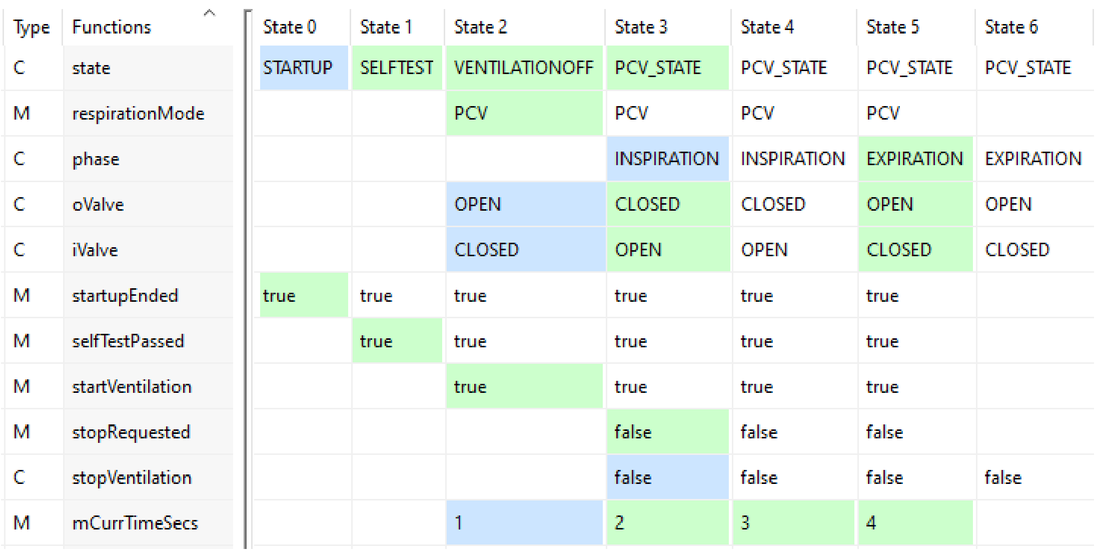
modeled by extending the rules `r_runPCVInsp` and `r_runPCVExp` as shown in Listing 5.11, and `r_runPSVInsp` and `r_runPSVExp` as shown in Listing 5.12. In this way, when the MVM is in expiration and detects, after an instant of time (a trigger window here modeled with the timer *timerTriggerWindowDelay*), a sudden pressure drop below the trigger sensitivity threshold (monitored function *dropPAW_ITS* - Listing 5.11 at line 15 and Listing 5.12 at line 13), the ventilator moves directly to the inspiration phase. On the other hand, the transition from inspiration to expiration is automatically performed when the inspiratory pressure goes beyond the maximum threshold set by the doctor (function *pawGTMaxPinsp* - Listing 5.11 at line 5 and Listing 5.12 at line 6).

Refinement proof

As already explained for the e-Pix case study, the automatic refinement proof has been carried out using the `AsmRefProver` tool. While for the e-Pix some internal activity was managed in a different way from one refinement level to the other, and thus it was necessary to add derived functions mapping refined behavior on the abstract one, for the MVM this was not necessary. In fact, refinements have been performed in a way that a specific function has always been implemented completely at a single refinement level.

Validation

With validation activities, the user can analyze the ASMETA specification using the simulator `AsmetaS`, the animator `AsmetaA`, and the model advisor `AsmetaMA`. Fig. 26 shows some simulation step using the animator `AsmetaA`. After completing the startup and self-test, the ventilator is in the *ventila-*



Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6
C	state	STARTUP	SELFTEST	VENTILATIONOFF	PCV_STATE	PCV_STATE	PCV_STATE	PCV_STATE
M	respirationMode			PCV	PCV	PCV	PCV	
C	phase				INSPIRATION	INSPIRATION	EXPIRATION	EXPIRATION
C	oValve			OPEN	CLOSED	CLOSED	OPEN	OPEN
C	iValve			CLOSED	OPEN	OPEN	CLOSED	CLOSED
M	startupEnded	true	true	true	true	true	true	
M	selfTestPassed		true	true	true	true	true	
M	startVentilation			true	true	true	true	
M	stopRequested				false	false	false	
C	stopVentilation				false	false	false	false
M	mCurrTimeSecs			1	2	3	4	

Figure 26: Simulation steps of the PCV mode with the animator `AsmetaA` for the MVM case study.

tion off state. In this case, as expected, the input valve is closed and the output valve is opened. When the start ventilation command is sent to the ventilator, and the PCV mode is selected, the ventilation starts in PCV mode with the inspiration phase, and the valves are moved to the expected position: the input valve is opened and the output valve is closed. After the inspiration duration, the ventilator moves to the expiration phase: the input valve is closed while the output valve is opened.

Scenario-based testing

In the scenario-based testing activity, the behavior of the MVM has been checked against the expected one by simulating all the possible states and transitions between them. As introduced in Sect. 4.3.2, the scenarios have been written in the `Avalla` language and executed using the validator `AsmetaV`. This tool checks if, at each step, the machine runs as expected by using the `check` commands.
An excerpt of a scenario for the MVM case study is shown in Listing 5.13. In this scenario, after having succeeded in the startup and self test, the MVM starts to ventilate in PCV mode. The main purpose of the scenario in the example is to verify that the valves are correctly set during inspiration and expiration.

Property verification

Once the validation and scenario-based testing have been performed, the property verification activity has been carried out. In this case study, LTL (Linear Temporal Logic) properties have been used. In particular, through the refinement steps, the properties in Tab. 14 have been written, translated into SMV, and verified. Note that the property "*Valves are never both open or closed at the same time*" has ben changed from level 0 and 1 (first row of the table) to the next ones (last row of the table). In

```
check state = STARTUP;
set startupEnded := true;
step
check state = SELFTEST;
set selfTestPassed := true;
step
check state = VENTILATIONOFF;
set startVentilation := true;
set respirationMode := PCV;
step
check state = PCV_STATE;
check oValve = CLOSED;
check phase = INSPIRATION;
check iValve = OPEN;
```

```
check state = PCV_STATE;
check oValve = CLOSED;
check phase = INSPIRATION;
check iValve = OPEN;
step
check state = PCV_STATE;
check oValve = OPEN;
check phase = EXPIRATION;
check iValve = CLOSED;
step
check state = PCV_STATE;
check oValve = OPEN;
check phase = EXPIRATION;
check iValve = CLOSED;
```

Listing 5.13: Example of `Avalla` scenario for the MVM case study in PCV mode.

Level	Property Description	SMV Property
0, 1	Valves are never both open or closed at the same time.	not **f**(iValve=oValve)
0, 1, 2, 3	When ventilation is off, the output valve is open and the input valve is closed.	**g**(state=VENTILATIONOFF **implies** (iValve=CLOSED and oValve=OPEN))
2, 3	Valves can be both closed when the MVM is in inspiratory or expiratory pause.	**g**(((phase=INPAUSE or phase=EXPAUSE) and (state = PCV STATE or state = PSV STATE)) **implies** (iValve=CLOSED and oValve=CLOSED))
2, 3	Valves are never both open or closed outside the inspiratory and expiratory pauses.	**g**((iValve=CLOSED and oValve=CLOSED) **implies** ((not ((phase=INSPIRATION or phase=EXPIRATION or phase=RM) and (state = PCV STATE or state = PSV STATE)))) or (not (state = VENTILATIONOFF or state = STARTUP or state = SELFTEST)))

Table 14: Properties verified for the MVM case study.

fact, after the inspiratory and expiratory pauses have been introduced, the first property has become too general and does not hold anymore.

C++ code generation

As in the e-Pix case study, for the MVM case study, the C++ code has been generated starting from the ASMETA specification thanks to the functionalities offered by the `Asm2C++` tool. The system analyzed is a prototypical version of the real MVM, and it is built using Arduino with the following components:

- Arduino Uno, which executes the state machine;
- 3 LEDs used to communicate the status of the input and output valves and the apnea alarm;
- A 1602 LCD display, which shows the current state;

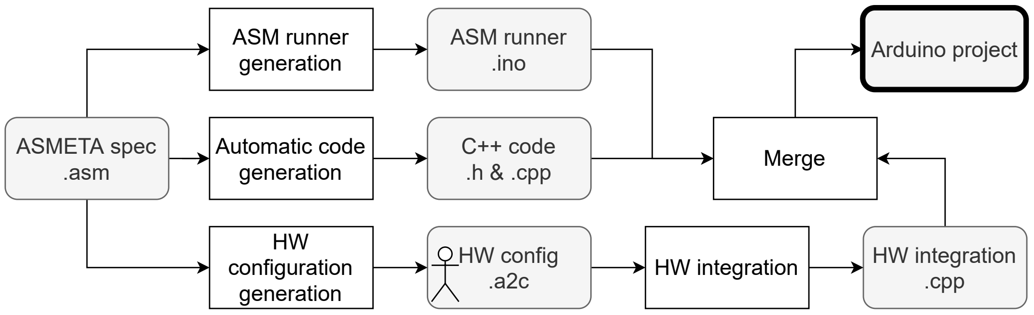


Figure 27: C++ code generation process.

```
[...]
void MVMController::r_runPCVInsp(){
  if (!stopVentilation[0]){ ... }
  if (expired(timerInspirationDurPCV)){
    if ((respirationMode == PCV)){
      if (cmdInPause){
        r_InPause();
      } else if (cmdRm){
        r_rm();
      } else {
        r_PCVStartExp();
      }
    }
  } else if (pawGTMaxPinsp)
    r_PCVStartExp();
}
```

```
void MVMController::r_runPCVExp(){
  if (stopVentilation[0]){
    r_stopVent();
  } else if (stopRequested){
    r_stopVent();
  } else if (expired(timerExpirationDurPCV)){
    if (cmdExPause){
      r_exPause();
    }else{
      r_PCVStartInsp();
    }
  } else if (expired(timerTriggerWindowDelay) &
        dropPAW_ITS){
    r_PCVStartInsp();
  }
} [...]
```

Listing 5.14: Example of the `.cpp` file for the MVM.

- 9 buttons, which simulate all the monitored functions contained in the ASM, namely the functions `dropPAW_ITS`, `pawGTMaxPinsp`, `cmdRm`, `cmdInPause`, `cmdExPause`, `flowDropPSV`, `respirationMode`, `stopRequested`, `startupEnded`, `selfTestPassed`, and `startVentilation`. They represent both user input and external breathing events.

The process followed to obtain the code to embed in the Arduino device is depicted in Fig. 27.

Starting from the last model refinement, the `Asm2C++` tool generates two different files: a `.h` and a `.cpp` file. They contain the translation of the ASM model as a C++ class, with its methods corresponding to the ASM rules. An example of the content of the `.cpp` file is reported in Listing 5.14, which reports the C++ translation of the two rules used for managing the PCV inspiration and expiration. Then, `Asm2C++` automatically generates an `.a2c` file, which is used for binding each ASM function to the physical pins of Arduino. It must be manually completed by the user, who has to insert the correspondence between Arduino physical pins, depending on the hardware configuration, and functions defined in the ASM model. An example of this file is reported in Listing 5.15: input and output valves are mapped on digital output pins, while the monitored functions are used to set if the current phase is finished or not (e.g., *startupEnded* and *selfTestPassed*) are read using digital input pins.

```
{
"arduinoVersion": "UNO",
"stepTime": 0,
"bindings": [
  {
    "mode": "DIGITALOUT",
    "function": "iValve",
    "pin": "D8"
  },
  {
    "mode": "DIGITALOUT",
    "function": "oValve",
    "pin": "D7"
  }, {
    "mode": "DIGITALIN",
    "function": "startupEnded",
    "pin": "A5"
  },
  {
    "mode": "DIGITALIN",
    "function": "selfTestPassed",
    "pin": "A4"
  },

  [...]
}
```

Listing 5.15: Example of the `.a2c` configuration file for the MVM.

```
#include "MVMController.h"

void MVMController::getInputs(){
  startupEnded = (digitalRead(A5) == HIGH);
  selfTestPassed = (digitalRead(A4) == HIGH);
  [...]
}
void MVMController::setOutputs(){
  if (iValve[0] != iValve[1]){
    if(iValve == OPEN)
      digitalWrite(8, LOW);
    else
      digitalWrite(8, HIGH);
  }
  if (oValve[0] != oValve[1]){
    if(oValve == OPEN)
      digitalWrite(7, LOW);
    else
      digitalWrite(7, HIGH);
  }
  [...]
}
```

Listing 5.16: Extract of the `hw.cpp` file containing hardware-specific functions.

After having manually completed the `.a2c` file with the correct mappings, `Asm2C++` generates two additional files: the `hw.cpp` and the `.ino`. The former implements the method related to the reading of inputs, `getInputs()`, and the one related to the writing of outputs, `setOutputs()` (see Listing 5.16). The latter contains the execution policy allowing one to run the ASM on Arduino. It cyclically performs four operations:

- `getInputs()`, which reads the inputs through digital and analog pins;
- `r_main()`, which represents the main rule of the ASM and executes all the rule allowing the changes of state;
- `setOutput()`, which sends the output values through the physical Arduino pins to the output components;
- `fireUpdateSet()`, which updates the values of controlled functions to be used in the next state.

The complete simulation, with the code generated by `Asm2C++` and uploaded to the Arduino-based circuit (see Fig. 28), has been executed by simulating the patient with its digital twin based on a simple lung model [54]. A complete simulation example is available at the following link: `https://youtu.be/a3fhqLpYVMI`.

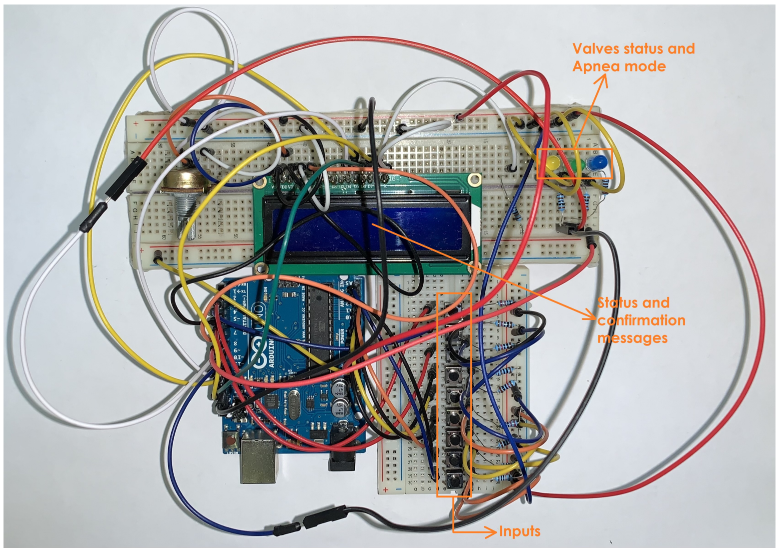

Valves status and Apnea mode
Status and confirmation messages
Inputs

Figure 28: The Arduino version of the MVM.

5.4 Model-based Testing with ASMETA

In the previous section, I have presented how a system can be developed from scratch, starting from its ASMETA specification. However, in some cases, the final implementation of the system may be already available and testers only need a guide on how to test the system, or simply need a way to automatically generate tests. For this purpose, model-based testing (MBT) is usually adopted. It consists of writing a formal model of the system and, then, deriving from it a test suite composed of abstract tests that have to be concretized in order to be executable on the real system.

As introduced in Sect. 4.4.1, ASMETA integrates the `ATGT` tool, which performs model-based test generation by exploiting the NuSMV model checker. In particular, `ATGT` builds some *test predicates* representing particular conditions that must be covered in order to satisfy the following coverage criteria:

- *Basic rule*: it requires that for every rule r_i there exists at least one test sequence for which r_i fires at least once, and there exists at least one test sequence for which r_i does not fire at least once.
- *Complete rule*: it requires that for every conditional rule r_i, the guard is true in at least one state of a test sequence and an update performed by r_i is not trivial.
- *Update rule*: it requires that for every function update $f := t$ there exists at least one test sequence for which the update is performed and it is not trivial.

- *Rule guard*: it requires that for every rule there exists a test in which the rule does not fire and the value v of some location that would be updated by the rule to v_r is different from the value it would be updated to in case the rule had fired.
- *MCDC*: it requires that every guard in every rule is tested according to the (masking) MCDC criterion.
- *Combinatorial interaction*: it requires that for every t-tuple of monitored locations (with limited domain), every combination of their possible values is tested in at least one state in a test sequence.
- *All criteria*: it requires all the above criteria.

In the following, I present the application of the model-based testing approach to two different medical systems, namely, the MVM (in its original version, which is currently running on the real devices marketed, developed using the Yakindu SCT tool) and the PHD protocol. In particular, for each case study, I report the phases of modeling, test generation, test concretization, and test execution with coverage evaluation.

5.4.1 Applying MBT to the MVM case study

As presented in Chapter 2, an important part of the MVM, now used and sold worldwide, is its controller, which receives operator inputs from the GUI, communicates with valve controllers, serial interfaces, and other sub-components, and sends them commands. It has been developed mainly using the Yakindu SCT tool, which allows users to draw the state machine that represents the behavior of the system and automatically generate the C++ code. In this section, I present how MBT activities based on the ASMETA framework can be carried out in order to test the Yakindu implementation of the MVM state machine. The process workflow is depicted in Fig. 29 and is analyzed below.

Modeling and V&V

The modeling activity has been performed as presented in Sect. 5.3.2. All models have been checked with the V&V activities previously described. Performing exhaustive and correct V&V is important, since generating tests from a faulty model may lead testers to think that the implementation is incorrect, while, indeed, it is not.

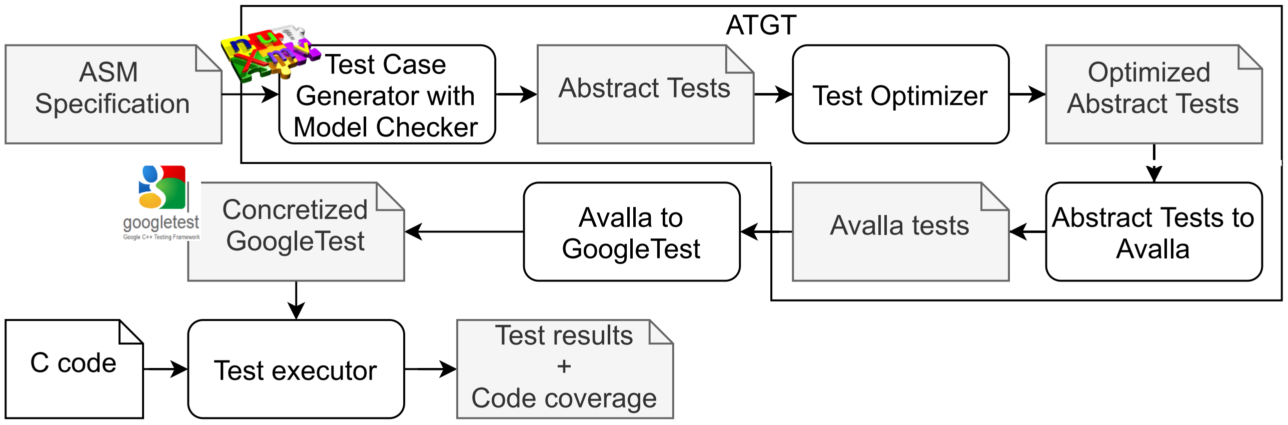


Figure 29: MBT process for the MVM case study.

Criteria	#Tps	Timeout 10 minutes #Tests	#Time-outs	Generation time [min]	#Tps covered	%Tps covered	Timeout 40 minutes #Tests	#Time-outs	Generation time [min]	#Tps covered	%Tps covered
Basic rule	72	13	29	345	43	60%	24	11	773	61	85%
Complete rule	2	0	0	0	2	100%	0	0	0	2	100%
Rule guard	124	1	60	601	64	52%	1	27	1080	97	78%
Rule update	89	0	52	520	37	42%	0	25	1000	64	72%
MCDC	148	10	55	581	93	63%	9	24	997	124	84%
2-Wise	420	77	0	1	420	100%	73	0	1	420	100%
All criteria	853	101	196	2048	659	77%	107	87	3852	768	90%

Table 15: Comparison between different criteria for automatic test cases generation.

Test generation

The test generation process is carried out using the ATGT tool. It starts from the ASM specifications, which have been validated and verified, and exploits the NuSMV counterexample generation.

In this case study, the abstract tests are stored in Avalla format. Furthermore, test generation has been executed using the *monitoring* optimization: when a test sequence ts is generated for a test predicate that has not yet been covered, the algorithms check if ts accidentally covers other test predicates and skips the other test predicates already covered. In this way, the test generation process is sped up.

Note that test generation using the model checker can be a time-expensive activity, even if monitoring is used. For this reason, a timeout has been added: for every test predicate tp to be covered, the model checker is interrupted if it reaches the timeout before producing a test, either because the test that covers tp exists but the model checker is unable to find it, or because tp is unfeasible, i.e., there is no test that covers it and the trap property is actually true. Unfortunately, ATGT is not able to distinguish the two cases by proving the unfeasibility of the test predicates.

Tab. 15 reports the comparison in terms of test predicates, number of generated tests, number of timeouts (or infeasible predicates), generation time, and number of test predicates covered for each coverage criterion using two different timeouts of 10 and 40 minutes. The results obtained confirm that

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;   ...
set respirationMode := PCV;   ...
step
check apneaAlarm = false;
check iValve = CLOSED;   ...
set respirationMode := PCV;   ...
step
check iValve = OPEN;   ...
```

Listing 5.17: Original scenario.

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;   ...
set respirationMode := PCV;   ...
step   ...


set respirationMode := PCV;   ...
step
check iValve = OPEN;   ...
```

Listing 5.18: Check Opt.

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;   ...
set respirationMode := PCV;   ...
step   ...



step
check iValve = OPEN;
```

Listing 5.19: Set Opt.

the higher the timeout, the higher the total number of covered test predicates (from 77% with timeout 10 minutes to 90% with timeout 40 minutes). The same behavior can be observed for the total number of tests generated and the generation time. Each coverage criterion, from 10 to 40 minutes of timeout, increases the number of tps covered except for *Complete Rule* and 2-Wise. The first does not generate any tests, because of monitoring optimization, since all tps are already covered by tests generated with *Basic Rule*. The second covers all the available tps for both values of timeout. Furthermore, for *2-Wise*, there is a small difference in terms of the number of tests generated (which is greater with a shorter timeout). This is because of the lower timeout, fewer tests are generated before trying to cover *2-Wise*, so more tps will result uncovered, and they will need more tests.

Test optimization

After having generated abstract tests, before their concretization, there may be the necessity to optimize them. Optimizations do not change the semantics of the tests, but improve the readability and translatability of the abstract tests to the concrete ones. In the experiments presented in this section, the following optimizations have been applied:

1. *Check optimization*: it removes unchanged controlled locations. In fact, if a controlled location in state s_i has not changed w.r.t. state s_{i-1}, the corresponding `check` is useless and can be removed, if present. For example, in Listing 5.17, the `check` command on the location *iValve* is repeated, so it is possible to remove the second one (see Listing 5.18).
2. *Set optimization*: it aims at removing `set` commands on monitored variables in state s_{i-1} if they are not actually asked to compute the update set for state s_i. For this reason, this optimization is based on automatically processing the scenario and keeping the set commands only if the values of the set functions are actually used by the ASM simulator to compute the update set. For example, in Listing 5.18, the second instance of **`set`** `respirationMode = PCV` is removed in Listing 5.19 since it is useless.

From the experiments carried out, the average number of `check` and `set` commands per state in the scenarios generated without optimization is, respectively, 37 and 15. By applying the check optimization technique, the optimized scenarios have an average of 11.22 check per state, while by applying the set optimization the average of set per state became 3.31. This reduction in commands allows for a consistent reduction in the length of each test case and, thus, in the time required for testing systems. In addition, having shorter test cases facilitates the task of pinpointing possible bugs.

Test concretization

To make abstract tests executable in Yakindu, they have been concretized using the `GoogleTest` framework. The code that converts scenarios to GoogleTests is available online at: `https://github.com/asmeta/mvm-asmeta/tree/master/mvm-scenario-converter`. In particular, the concretization process consists in the following three consecutive steps:

1. *Mapping of ASMETA functions to state machine variables*: The first step for tests concretization is the mapping of ASMETA functions to state machine variables. For this purpose, a JSON configuration file is automatically generated and filled with all the functions set or checked in the Avalla scenarios. Then, the mapping has to be performed manually by the user. For each function, the JSON file contains:

   - `asmName`, which is the name that the function assumes in the ASMETA specification;
   - `cName`, i.e., the name of the function in the C++ code;
   - `commandType`, specifying the type of function, chosen between IN_EVENT (representing events raised by the user), VAR (representing internal fields of the state machine), OPERATION (representing function interacting with hardware components), STATE (representing the state of the state machine), and TBD (the default type, TBD functions are ignored during test concretization, but are used only in the ASMETA model).

   An example of the content of the JSON file is reported in Listin 5.20. The function *startVentilation* is IN_EVENT since it is raised by the user. *mode* is VAR because it represents an internal field of the state machine, and *iValve* is an OPERATION function, because it interacts with hardware components, i.e., the input valve. Functions used only in the ASMETA model but not in the C++ code (e.g. *time*) are set to be ignored (TBD type).

2. *Hardware mocking*: Considering that the MVM state machine directly interacts with hardware, during test concretization, it has to be mocked. For this reason, mocking classes have been written using the same interface as the real classes of hardware components. Then, the mocking file

```
[ { ”asmName”: ”startVentilation”,
        ”cName”: ”startVentilation”,
        ”commandType”: ”IN_EVENT”
},{ ”asmName”: ”time”, ”cName”: ”time”,
        ”commandType”: ”TBD”
},{ ”asmName”: ”iValve”,
        ”cName”: ”defaultMock−>getInValveStatus”,
        ”commandType”: ”OPERATION”
},{ ”asmName”: ”state”, ”cName”: ”state”,
        ”commandType”: ”STATE”
},{ ”asmName”: ”mode”, ”cName”: ”mode”,
        ”commandType”: ”VAR”}]
```

Listing 5.20: JSON file for function mapping.

```
set mode := PSV;
set startVentilation := true;
step
check time = 3;
check oValve = CLOSED;
check iValve = OPEN;
check state =
    MAIN_REGION_PSV_R1_INSPIRATION;
```

```
sm−>setMode(PSV);
sm−>raiseStartVentilation();
runner−>proceed_time(100);

EXPECT_EQ(valveMock−>getOutValveStatus() , CLOSED);
EXPECT_EQ(valveMock−>getInValveStatus() , OPEN);
EXPECT_TRUE(sm−>isStateActive(
     MAIN_REGION_PSV_R1_INSPIRATION));
```

Listing 5.21: Test concretization from an `Avalla` scenario fragment to a GoogleTest test case.

is automatically included in the test suite by the scenario concretization process. The complete mocking file for the MVM case study is available online at `https://github.com/asmeta/mvm-asmeta/blob/master/mvm-scenario-converter/additional_files/mock.c`.

3. *GoogleTest code generation*: After having mocked the hardware and configured the mapping between ASMETA and C++ files, the concretization can be completed by generating a GoogleTests test suite. Considering that the MVM has been developed as a cycle-based state machine, with a cycle duration of 100ms, the translation of the `step` `Avalla` command has been done by replacing it with the Yakindu command `proceed_time(100)`. The other Avalla commands are concretized as explained in Tab. 16.

   Listing 5.21 shows a test concretization example of an `Avalla` scenario. In particular, IN_EVENT functions (such as *startVentilation*) are raised only when they are set to true in the Avalla scenario, while VAR functions, such as *mode*, are set in the GoogleTest test case when there is a corresponding set in the `Avalla` scenario and checked when there is a corresponding check in the scenario. OPERATION functions, such as *iValve*, are converted into method

Function type	Set	Check
STATE	`//`	`EXPECT_TRUE(sm->isStateActive([stateName]))`
IN_EVENT	`sm->raise[cName]()`	`//`
VAR	`sm->set[cName]([value])`	`EXPECT_EQ(sm->get[cName](),[value])`
OPERATION	`[cName]([value])`	`EXPECT_EQ([cName](),[value])`

Table 16: Translation rules between Avalla and GoogleTest instructions (`sm` is the generic name used to indicate the state machine object in Yakindu).

	Timeout 10 minutes			Timeout 40 minutes		
Criteria	**Statement Cov.**	**Branch Cov.**	**Function Cov.**	**Statement Cov.**	**Branch Cov.**	**Function Cov.**
Basic rule	65.69%	63.48%	59.46%	80.97%	81.32%	79.05%
Complete rule	65.69%	63.48%	59.46%	80.97%	81.32%	79.05%
Rule guard	66.19%	63.91%	60.47%	81.48%	81.74%	80.07%
Rule update	66.19%	63.91%	60.47%	81.48%	81.74%	80.07%
MCDC	70.24%	69.85%	65.54%	81.48%	81.74%	80.07%
2-Wise	70.24%	69.85%	65.54%	81.98%	82.17%	81.08%
All criteria	70.24%	69.85%	65.54%	81.98%	82.17%	81.08%

Table 17: Coverage reached using different timeouts and coverage criteria.

calls. If there is a check in the `Avalla` scenario on an OPERATION function, a check is also performed in the GoogleTest test case. Finally, the STATE function represents the active state of the machine.

Test execution

Having concretized the abstract tests, they have been used for testing the C++ code of the MVM controller. Tab. 17 reports the incremental coverage, in terms of statements, branches, and functions, reached by the tests automatically generated using 10 and 40 minutes of timeout. The results obtained confirm the observations made after Tab. 15: increasing the timeout leads to an increment of the covered test predicates and, consequently, of the code coverage. Note that the coverage in Tab. 17 increased from one criterion to the next, except for *Complete rule* and *Rule update*, for which no significant test is generated. This means that no criterion could have been skipped, since all the criteria contribute to increasing the coverage.

Even when the higher timeout is used, full coverage of the MVM code has not been obtained. This is reasonable since, starting from the code automatically generated by Yakindu SCT, many parts of the code are only used by Yakindu itself and cannot be mapped to external calls. However, the coverage reached in this case is higher than that reached during the development of the MVM itself. In fact, unit testing of the MVM controller was not mandatory in order to obtain the safety certification since the component was not classified in class C (see Sect. 1.2.1). For this reason, only a few tests were written manually, and those obtained with the process described in this section are an important asset, both for usability testing and integration/system testing that, indeed, is mandatory for each medical device, regardless of its safety class.

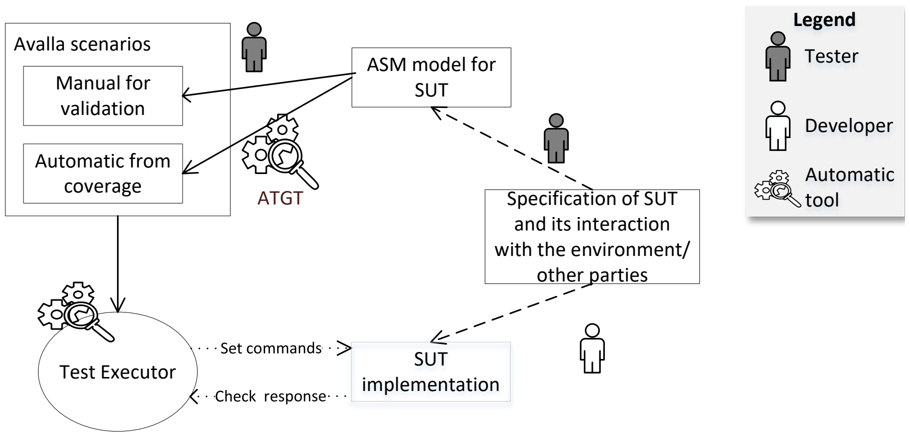


Figure 30: MBT process for the PHD case study.

5.4.2 Applying MBT to the PHD protocol case study

In this section, I present how MBT activities based on the ASMETA framework can be carried out in order to test the manager part of Antidote 2.1[3], an open-source implementation of the PHD protocol (previously presented in Sect. 5.2.2). The Antidote source code is written in C and composed of the following source folders: *api*, *asn1*, *communication*, *dim*, *resources*, *specializations*, *trans*, and *util*. In order to focus the testing process solely on the manager, the functionalities contained in the *communication* module have been further split into two folders: *agent* and *manager*.

The process workflow is presented in Fig. 30. As previously done for the MVM case study, the ASMETA specification is derived from the system requirements specification. Then, using the `ATGT` tool, abstract tests are derived in the `Avalla` format. Normally, abstract tests need to be concretized. However, in this case, Antidote can be tested directly with a test executor capable of interpreting `Avalla` files, so the test concretization and test execution phases coincide.

Note that, in this case study, the approach is different from the one presented for the MVM. In fact, for the MVM a refinement approach has been used for writing models and, then, the tests have been generated from the last refinement level. Instead, for the PHD case study, I present an alternative approach called `RATE` (Refinement And Test Execution), which is based on alternating refinement, test generation, and test execution in order to gather, from the coverage reached, information about the parts in the code that are missing in the model. In this way, by using a gray-box approach, the model can evolve to be more adherent to the implementation. Furthermore, this approach can be used when reverse engineering is needed to derive requirements from the implementation.

Data useful for replicating the results shown in this section are available online at `https://github.com/asmeta/RATE/tree/main/Case_studies/PHD_Protocol`.

[3]`https://github.com/signove/antidote`

Modeling

In the following, I present 6 refinement steps that have been made in the case study of the PHD protocol. For each of them, simulation, animation, scenario-based testing, refinement proof, and verification activities have been performed, in order to verify the correctness of the behavior modeled w.r.t. the official specification [102]. The complexity of the refinement levels increases from one level to the other, as well as the number of rules contained in each model.

- **Level 0 - Main manager transitions**: In this first level, only three states have been modeled: *Disassociating*, *Unassociated*, and *Operating*. The transition from one state to the next one and the response depend on the current state and the message received. Listing 5.22 reports a fragment of ASM_0 written using the `AsmetaL` language. The signature of ASM_0 contains three functions: `status`, `transition`, and `message`. The `transition` function models the type of request to be sent to the manager, and is defined as a monitored function since its value can be controller by external factors or entities, e.g., by the agent. The `status` function represents the current state of the manager, while the `message` represents the response from the manager. Both functions are defined as controlled, since their value is managed by the ASM model.

- **Level 1 - Remote operation management**: In the first level of refinement, messages used for remote operation management (`rx_roiv`, `rx_rors`, `rx_rorj`) have been added. Moreover, since not all messages can be used in every state, an invariant for each state has been added, to guarantee that only messages valid for each state can be sent by the agent. This is a simplification that allowed testing the system incrementally, by isolating regular and exceptional behaviors. Listing 5.23 reports a fragment of ASM_1 written in the `AsmetaL` language, which extends the possible messages and introduces the invariants.

- **Level 2 - PHD configuration management**: This level of refinement adds the states modeling the exchange of configutation between devices working with the PHD protocol: *CheckingConfig* and *WaitingForConfig*. Furthermore, the transitions, messages, and rules related to the newly added states are modeled by ASM_2.

- **Level 3 - Error management**: All `rors` messages related to error management were not yet modeled by the previous level of refinement. Therefore, in the fourth model ASM_3, the `rors` message and its subtypes (`rors-*`) have been added. From the protocol specification, it can be noticed that these messages trigger a relevant part of the protocol between the states *Disassociating* and *Unassociated*, and within the states *Operating*, *CheckingConfig*, and *WaitingForConfig*.

```
asm PHD0
import StandardLibrary

signature:
  // DOMAINS
  enum domain Status = {UNASSOCIATED, OPERATING, DISASSOCIATING}
  enum domain Transition = {REQ_ASSOC_REL, REQ_ASSOC_ABORT, RX_RLRE, RX_ABRT, RX_AARQ,
       RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION, RX_AARE,
       RX_RLRQ}
  enum domain Message = {MSG_NO_RESPONSE, MSG_RX_AARE, MSG_RX_ABRT, MSG_RX_RLRQ,
       MSG_RX_RLRE}

  // FUNCTIONS
  controlled status: Status
  monitored transition: Transition
  controlled message: Message

definitions:
  rule r_Unassociated =
    switch transition
      case REQ_ASSOC_REL:
        par
          status := UNASSOCIATED
          message := MSG_NO_RESPONSE
        endpar
      case REQ_ASSOC_ABORT:
          [...]
    rule r_Operating =
    switch transition
      [...]
  rule r_Disassociating =
    switch transition
      [...]

main rule r_Main = par
    if (status = UNASSOCIATED) then
    r_Unassociated[]
  endif
  if (status = OPERATING) then
    r_Operating[]
  endif
  if (status = DISASSOCIATING) then
    r_Disassociating[]
  endif
endpar

// INITIAL STATE
default init s0:
  function status = UNASSOCIATED
```

Listing 5.22: ASMETA specification of ASM_0, specifying the main PHD manager transitions.

```
asm PHD1
import StandardLibrary

signature:
  // DOMAINS
  [...]
  enum domain Transition = {REQ_ASSOC_REL, REQ_ASSOC_ABORT, RX_RLRE, RX_ABRT, RX_AARQ,
      RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION, RX_AARE, RX_RLRQ, RX_ROIV,
      RX_ROER, RX_RORJ}
  enum domain Message = {MSG_NO_RESPONSE, MSG_RX_AARE, MSG_RX_ABRT, MSG_RX_RLRQ,
      MSG_RX_RLRE, MSG_RX_PRST}
  [...]

definitions:
  [...]

  INVAR (status = DISASSOCIATING) implies (transition = RX_RLRE or transition = REQ_ASSOC_ABORT
    or transition = RX_AARQ or transition = RX_AARE or transition = RX_RLRQ
    or transition = REQ_ASSOC_REL or transition = RX_ABRT or transition = RX_ROIV
    or transition = RX_ROER or transition = RX_RORJ)

  INVAR (status = OPERATING) implies (transition = REQ_ASSOC_REL or transition = REQ_ASSOC_ABORT
    or transition = RX_AARQ or transition = RX_AARE or transition = RX_RLRQ or transition = RX_RLRE
    or transition = RX_ABRT or transition = RX_ROER or transition = RX_RORJ or transition = RX_ROIV)

  INVAR (status = UNASSOCIATED) implies (transition = REQ_ASSOC_REL or transition = REQ_ASSOC_ABORT
    or transition = RX_AARE or transition = RX_RLRQ or transition = RX_RLRE
    or transition = RX_ABRT or transition = RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION)

  [...]
```

Listing 5.23: ASMETA specification of ASM$_1$, introducing the remote operation management.

Moreover, this level of refinement introduces two particular sequences of transitions in the model, which, according to the protocol specification [102], have to be managed differently:

1. When the manager is in the *WaitingForConfig* state and receives from the agent a message `rx_roiv_confirmed_event_report`, it should move to the *CheckingConfig* state. However, the internal handling of this transition is different depending on whether the state *WaitingForConfig* was entered, with a transition from the state *Unassociated* or from the state *CheckingConfig*. In the former case, no configuration similar to the one transmitted by the agent is present in the manager pool of configurations, and so, additional functions that ask for configuration attributes are used. In the latter case, a similar configuration was previously transmitted, and thus the configuration is already in the memory of the Antidote manager.
2. The behavior of the message `rx_roiv_confirmed_event_report`, when causing a loop in the *CheckingConfig* state, is different if executed right after another same message that brought the manager from the state *WaitingForConfig* to the *CheckingConfig* state. In this case, additional functions handling the new measurement received from the agent are executed.

```
asm PHD4
import StandardLibrary

signature:
  // DOMAINS
  [...]
  enum domain Transition = {REQ_ASSOC_REL, REQ_ASSOC_ABORT,
       RX_AARQ_ACCEPTABLE_AND_KNOWN_CONFIGURATION,
       RX_AARQ_ACCEPTABLE_AND_UNKNOWN_CONFIGURATION,
       RX_AARQ_UNACCEPTABLE_CONFIGURATION, RX_AARE, RX_RLRQ,
       RX_RLRE, RX_ABRT, RX_ROIV_CONFIRMED_EVENT_REPORT, RX_ROIV, RX_ROER, RX_RORJ,
       REQ_AGENT_SUPPLIED_UNKNOWN_CONFIGURATION, RX_RORS,
       REQ_AGENT_SUPPLIED_KNOWN_CONFIGURATION,
       RX_RORS_CONFIRMED_ACTION, RX_RORS_CONFIRMED_SET, RX_RORS_GET, RX_AARQ,
       RX_AARQ_INVALID, RX_AARQ_EXTERNAL}
  [...]

definitions:
  [...]
```

Listing 5.24: ASMETA specification of ASM$_4$, introducing protocol and configuration management.

- **Level 4 - Protocol and configuration management**: During connection, an agent may try to use a wrong `protocol-id` or with an unknown (or external) configuration, which is identified with a specific `protocol-id` value (0xFFFF). For this reason, as shown in Listing 5.24, ASM$_4$ models the message `rx_aarq` with two additional variants, respectively, with an invalid `protocol-id` and an external `protocol-id`.

- **Level 5 - Invalid messages management**: This refinement level removes the invariants previously defined in order to limit the messages received by the manager to those valid. In fact, the official IEEE specification [102] of the PHD requires managers to deal with invalid messages, too, and, when a manager receives a message that is not defined in its current state, it must reply with an abort message.

- **Level 6 - Invalid invoke-id management**: One of the aspects that were not yet captured by the ASM specification was the dependence of the manager's behavior based on the `invoke_id` contained in each APDU. For example, if the manager is in *WaitingForConfig* and receives `rors-*`, `roer`, or `rorj` messages, it can produce `no response` if `invoke_id` is valid or an abort otherwise. To manage this particular behavior, as shown in Listing 5.25, ASM$_6$ has an additional monitored function `invokeIdValid` that is combined with the `transition` function to establish if the message must be sent with valid or invalid `invoke_id`.

Test Generation

Starting from the ASM specification, tests have been generated using the criteria presented in Sect. 5.4 for each refinement level. For the *Combinatorial interaction* criteria, $t = 2$ and $t = 3$ have been used.

```
asm PHD6
import StandardLibrary

signature:
  // DOMAINS
  [...]

  // FUNCTIONS
  controlled status: Status
  monitored transition: Transition
  controlled message: Message
  monitored invokeIdValid: Boolean
  [...]

definitions:
  [...]
  rule r_Waiting_For_Config =
    switch transition
      [...]
      case RX_ROER:
          if invokeIdValid = true then
            par
              status := WAITING_FOR_CONFIG
              message := MSG_NO_RESPONSE
            endpar
          else
            par
              status := UNASSOCIATED
              message := MSG_RX_ABRT
            endpar
          endif
          [...]
```

Listing 5.25: ASMETA specification of ASM_4, introducing protocol and configuration management.

However, since *2-Wise* and *3-Wise* are performed only considering the monitored functions, only with the last refinement level (which has two monitored functions), the former is effective, while the latter has not produced any test predicate. More details on the test sequences obtained, their length and the number of steps are reported in Tab. 18. Furthermore, in order to compare sequences automatically generated by `ATGT` and manual testing, manually written `Avalla` scenarios have also been used.

Test concretization and execution

As previously mentioned, for the PHD protocol case study, the test executor is able to deal with `Avalla` files, so the test concretization and test execution phases coincide. In particular, at every step, the test executor sends a suitable message to the Manager, checks the response, and the target state. To perform the test execution, the ProTest tool, originally presented in [182] and modified for dealing with `Avalla` files, has been used. It acts as an agent by interacting with the manager implementation that is executed on the server side. It builds the messages, sends them to the manager, and checks the conformance of the response received from it. Moreover, it integrates gcov[4] which allows users to evaluate the coverage of the performed tests, and to check which statements are not covered.

[4]`https://gcovr.com`

Refinement	Test generation strategy	Test sequences					Code coverage			# Test predicates
		#	steps							
		sequences	min	max	total	avg	statement	function	branch	
ASM_0	*Basic rule*	21	1	4	62	2.95	34.1%	**54.3%**	22.8%	45
	Complete rule	3	1	4	8	2.67	32.6%	51.8%	23.2%	3
	Rule update	21	1	4	64	3.05	**35.6%**	**54.3%**	23.8%	42
	Rule guard	22	1	4	66	3.00	**35.6%**	**54.3%**	24.1%	66
	MCDC	21	1	4	62	2.95	**35.6%**	**54.3%**	24.6%	48
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	24	1	4	71	2.96	**35.6%**	**54.3%**	24.1%	204
	manual	1	34	34	34	34.00	**35.6%**	**54.3%**	**27.0%**	//
ASM_1	*Basic rule*	27	1	4	83	3.07	42.1%	**67.1%**	27.8%	57
	Complete rule	3	1	4	8	2.67	33.2%	53.0%	21.8%	3
	Rule update	27	1	4	85	3.15	**43.6%**	**67.1%**	**29.2%**	54
	Rule guard	27	1	4	85	3.15	**43.6%**	**67.1%**	**29.2%**	84
	MCDC	27	1	4	83	3.07	**43.6%**	**67.1%**	25.0%	60
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	31	1	4	95	3.06	**43.6%**	**67.1%**	**29.2%**	258
	manual	1	46	46	46	46.00	**43.6%**	**67.1%**	**29.2%**	//
ASM_2	*Basic rule*	56	1	4	182	3.25	58.2%	76.8%	37.5%	115
	Complete rule	5	1	4	15	3.00	39.5%	55.5%	25.3%	5
	Rule update	56	1	5	187	3.34	58.2%	76.2%	31.9%	110
	Rule guard	60	1	5	203	3.38	58.2%	76.2%	38.8%	170
	MCDC	56	1	4	182	3.25	**60.9%**	**77.4%**	**39.8%**	120
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	68	1	5	230	3.38	58.2%	76.2%	38.8%	520
	manual	5	11	33	104	20.80	58.2%	72.2%	39.0%	//
ASM_3	*Basic rule*	71	1	4	234	3.30	**63.5%**	**79.9%**	**42.6%**	141
	Complete rule	5	1	4	15	3.00	42.3%	61.0%	28.4%	5
	Rule update	69	1	5	235	3.41	60.8%	78.7%	41.6%	136
	Rule guard	75	1	5	259	3.45	60.8%	78.7%	41.6%	209
	MCDC	68	1	4	222	3.26	60.8%	78.7%	41.6%	146
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	86	1	5	298	3.47	60.8%	78.7%	41.6%	637
	manual	5	11	38	121	24.20	61.1%	78.7%	42.0%	//
ASM_4	*Basic rule*	69	1	4	222	3.22	63.9%	**79.9%**	**43.0%**	145
	Complete rule	5	1	4	15	3.00	39.8%	57.9%	25.3%	5
	Rule update	71	1	5	239	3.37	61.2%	78.7%	41.6%	140
	Rule guard	78	1	5	267	3.42	61.2%	78.7%	41.6%	215
	MCDC	69	1	4	222	3.22	61.2%	78.7%	42.0%	150
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	88	1	5	301	3.42	61.2%	78.7%	41.6%	655
	manual	5	14	38	124	24.80	**64.0%**	**79.9%**	**43.0%**	//
ASM_5	*Basic rule*	83	1	4	272	3.28	**64.0%**	**79.9%**	**43.1%**	157
	Complete rule	5	1	4	15	3.00	41.4%	59.1%	26.0%	5
	Rule update	82	1	5	277	3.38	61.3%	78.7%	42.2%	152
	Rule guard	90	1	5	307	3.41	61.3%	78.7%	42.2%	233
	MCDC	96	1	4	316	3.29	**64.0%**	**79.9%**	**43.1%**	294
	2-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	117	1	5	400	3.42	**64.0%**	**79.9%**	**43.1%**	841
	manual	5	15	41	135	27.00	**64.0%**	**79.9%**	**43.1%**	//
ASM_6	*Basic rule*	95	1	4	307	3.23	**64.2%**	**79.9%**	**43.8%**	169
	Complete rule	5	1	4	15	3.00	40.4%	59.8%	26.6%	5
	Rule update	93	1	5	310	3.33	**64.2%**	**79.9%**	**43.8%**	176
	Rule guard	103	1	5	347	3.37	**64.2%**	**79.9%**	**43.8%**	257
	MCDC	109	1	4	356	3.27	**64.2%**	**79.9%**	**43.8%**	318
	2-Wise	28	1	3	61	2.18	35.1%	54.3%	24.8%	44
	3-Wise	0	0	0	0	0.00	0.0%	0.0%	0.0%	0
	All criteria	134	1	5	446	3.33	**64.2%**	**79.9%**	**43.8%**	969
	manual	7	12	41	159	22.71	**64.2%**	**79.9%**	**43.8%**	//

Table 18: Coverage results for each refinement/test generation strategy applied to the PHD protocol case study. The coverage values in bold represent the highest coverage reached at that refinement level.

Tab. 18 presents the coverage reached for each level of refinement in the PHD case study with all testing criteria. It can be seen that the code coverage obtained with the tests generated from ASM_0 is very similar for all test generation strategies (around 35% for statement coverage, 54% for function coverage, and 23% for branch coverage) except for combinatorial-based methods, since, as mentioned before, no tests are generated by *2-Wise* and *3-Wise* criteria. In the first step of refinement (ASM_1) the coverage increases as expected, as well as for the one in ASM_2, because the ASM models have been improved by adding remote operation management and configuration management. For the other refinement levels, from ASM_3 to ASM_6, the coverage slowly increases since most of the main aspects of the protocol (which represent the majority of statements and functions in the Antidote code) were already captured by ASM_2. Note that full coverage has not been reached, even with the last level of refinement ASM_6. Analyzing the uncovered statements, it can be seen that they are mainly related to dead code (such as functions declared with an empty body or never used), or negative use cases (exceptions), often regarding internal configurations of the manager.

To evaluate the effectiveness of the proposed approach, manual testing has been performed with manually written `Avalla` scenarios. The main revealed differences between manual tests and generated tests are the average sequence length and the number of test sequences (see Tab. 18). In fact, when automatic test generation is used, test predicates are used to automatically generate test sequences, and a single sequence mainly covers a single test predicate. On the contrary, with manual tests, the length of test sequences is higher, but their number is lower: users tend to cover more test predicates in each scenario.

Even if the total number of sequences in manual testing is significantly lower than the other criteria (in some refinement level also more than 90% lower), the coverage is, in general, equal or only slightly lower than the automatic test generation with the highest coverage. On the basis of the outcomes and considering the great effort required for writing tests manually, it can be concluded that automatic test generation can substitute the manual tests since it guarantees the same coverage with lower effort by the user. Moreover, with manual tests, one can miss covering a specific behavior, make a mistake in test writing, or in defining the test oracle.

Finally, the proposed approach has been revealed to be successful, since by applying MBT guided by coverage information, as presented in this section, several faults have been found and fixed in a new release of the protocol[5]:

[5]`https://github.com/fmselab/antidote3`

- The official IEEE 11073-20601 specification requires the use of `rx_abrt` as response for the sequence "`Unassocciated` + `req_assoc_abort`". However, the tests generated from ASM_0 have highlighted that the original Antidote implementation used "`no response`" instead.

- The sequence "`checking_config` + `rx_aarq` → `rx_abrt`" caused a transition mismatch when executing tests derived from ASM_1. By checking the code, three transitions for sub-types of the event `rx_aarq_*` were implemented, but the case where the `rx_aarq` message is received in state `checking_config` was missing. This means that the manager did not respond to the message `rx_aarq` itself. Since the IEEE specification requires "`rx_abrt`" as a response when an unexpected message is received by the manager, the transition "`checking_config` + `rx_aarq` → `unassociated` + `rx_abrt`" has been added to the fixed version of the Antidote manager state table.

- The sequence "`disassociating` + `rx_rors` → `unassociated` + `rx_abrt`" caused a response mismatch when executing tests derived from ASM_3. Indeed, the answer given by the manager was "`no response`" if a valid invoke-id was provided, but the IEEE specification always requires "`rx_abrt`" as response for this message.

- The Antidote manager did not check the invoke-id contained into the "`rx_roer`" and "`rx_rorj`" messages. Indeed, the official IEEE specification requires "`rx_abrt`" as a response to these two messages when the invoke-id is invalid, and "`no response`" otherwise. The bug has been revealed while executing tests derived from the last refinement level (ASM_6).

In conclusion, the presented approach has allowed testers not only to decrease the effort required to test complex medical systems such as the PHD protocol, but also to identify some faults and conformance errors in the open-source Antidote implementation.

5.5 Conclusion

In this chapter, I have presented how the ASMETA framework can be easily applied to medical systems to increase their quality and reliability. In fact, as explained in Sect. 5.1, all activities required by the main standard for medical software certification can be easily mapped to the activities and functionalities offered by ASMETA.

The main directions in which the ASM-based development process and ASMETA in general can aid developers are the development of correct-by-construction code and model-based testing activities.

When there is no existing code, the developer can start modeling the system using the AsmetaL language. Then, he/she can verify and validate the modeled systems using several tools embedded in the ASMETA framework (AsmetaS, AsmetaA, AsmetaV, and AsmetaSMV). Finally, after the model has been verified, the developer can derive the code for the actual device from the ASM specification using the Asm2C++ tool. This process has been successfully applied and tested for two different medical case studies, namely the e-Pix and the MVM. The first is a smart pill box, which has been developed by a local company and is based on Arduino. By applying the ASMETA-based development process, we have been able to fully cover all the activities required by medical software certification authorities: starting from a formal specification, obtained through several refinements, we have proved the necessary safety properties of the device, and we have generated automatically the source code to be embedded on the Arduino. The second device, presented in detail in Chapter 2, is a mechanical ventilator, developed for COVID-19, I contribute to the developing and certification process. Even if the ASMETA process has not been applied during the actual development of the product, in this section I have shown how it could have been, in principle, applied for the development of such a complex medical device. As for the e-Pix case study, also in this case, starting from the formal specifications we have been able to obtain a correct-by-construction source code.
On the other hand, when the system code is already available and testers want guidance in generating test cases, a model-based approach can be chosen. In this case, after the tester has written the system specification using the AsmetaL language, and performed V&V activities on it, abstract tests can be derived thanks to the ATGT tool. The obtained tests can then be concretized and executed against the implementation of the medical system under test. This process has been applied to the MVM and to the PHD protocol case study. For the MVM case study, I have shown how one can apply model-based testing by generating, from the formal specification, test cases that can be concretized as Google Tests. These tests have been executed against the software which currently runs on the real MVM in production. In this way, we have been able to increase the code coverage w.r.t. the one we previously reached during the development of the device. Instead, for the PHD protocol case study (a protocol allowing the communication between medical devices) I have presented a different approach, called RATE: starting from the already existing source code, the ASMETA model has been obtained by identifying which part of the source code were not covered. With this approach, by looking which part were missing in the model and by implementing them in the model in a way compliant to the system specifications, a failing test represents behaviors wrongly implemented. Finally, by applying the ASMETA framework under different perspectives, in both cases, the coverage obtained has been satisfactory and, moreover, the applied process has allowed to discover bugs or conformance errors.

Chapter 6. Combinatorial testing for complex PEMS

In the previous chapter, the application of model-based techniques to PEMS case studies exploiting the ASMETA framework has been presented. However, testers may need to deal with complex medical systems that have several inputs and outputs, and testing them extensively may be unfeasible. Moreover, bugs may be revealed only by precise combinations of a few inputs. For these reasons, this chapter introduces the combinatorial testing strategy and shows how it can be applied to the MVM (previously presented in Chapter 2) and PHD protocol (previously analyzed in Chapter 5) case studies, to investigate how it can contribute in aiding testers while working with highly configurable and complex systems by reducing the number of test cases.

This chapter is based on the work presented in [31–35] and is structured as follows. Sect. 6.1 introduces the general concepts of combinatorial interaction testing, while Sect. 6.2 explores how it can be applied when sequences of inputs have to be tested in a combinatorial way. Sect. 6.3 presents a method, a set of heuristics, and a formula for comparing combinatorial generators when testers have to choose the best one for the specific application scenario. Finally, Sect. 6.4 tackles the advantages and limits of parallelizing combinatorial test generation combining multithread with Multi-valued Decision Diagrams or SMT solvers, and Sect. 6.5 concludes the chapter.

6.1 Introduction

Combinatorial interaction testing (CIT) has been an active area of research for many years and has proven to be very effective for testing complex systems, especially for those with several input parameters, such as medical systems.

Combinatorial testing is based on the idea that not every parameter contributes to every fault, and most faults are caused by interactions between a relatively small number of parameters [114]. Studies conducted by NIST from 1999 to 2004 showed that combinatorial testing is more efficient at detecting faults than conventional methods: it has proved to have a fault detection equal to exhaustive testing while reducing the test suite size 20 to 700 times. Suppose that you want to test a system with 7 switches that can be either ON or OFF. If exhaustive testing is used, $2^7 = 128$ test cases are needed. However, if one decides to test only the interaction between pairs of parameters (strength $t = 2$), 8 tests are enough. Furthermore, if constraints restricting input combinations are added to the system under test, the number of test cases may be even further reduced.

Combinatorial testing can be considered a particular type of model-based testing. In fact, when working with combinatorial testing, testers need to define a constrained combinatorial model in which the parameters, their bounds, and the constraints are defined.

Definition 1 (Constrained Combinatorial Model)**.** Let $P = \{p_1, ..., p_n\}$ be a set of n parameters, where every parameter p_i assumes values in the domain $D_i = \{v^i_1, \dots, v^i_{o_i}\}$. Let D be the set of all D_i, i.e., $D = \{D_1, \dots, D_n\}$ and $C = \{c_1, ..., c_m\}$ be the set of constraints over the parameters p_i and their values v^i_j. We say that $M = (P, D, C)$ is a *Constrained Combinatorial Model.*

Listing 6.1 shows an example of a constrained combinatorial model written in CTWedge [87] for the MVM case study (see Chapter 2). In particular, a combinatorial model reports:

- the list of *parameters*, which represent the inputs, outputs, or configurations of the system under test. In CTWedge, the parameters can be enumerative, Boolean, or integer ranges;
- the list of *constraints*, which are expressed as logical formulas, and limit the possible configurations.

A test can be *valid* if and only if it does not violate any constraint; otherwise, it is defined as *invalid*. The same concept of validity applies to tuples: a tuple tp is valid (or *feasible* or *coverable*) if there exists a valid test that covers tp. Otherwise, a tuple is invalid or unfeasible. For example, considering the model in Listing 6.1, the tuple

$$tp = \{NextState = OFF, PowerOff = false\}$$

is not valid since it violates the first constraint. After having defined what a combinatorial model is and what the validity of a test is, it is possible to define the concept of *valid test suite* as follows.

Definition 2 (Valid Test Suite)**.** Let $M = (P, D, C)$ be a constrained combinatorial model. Given a combinatorial test suite TS, we say that TS is *valid* if all tests $ts_i \in TS$ are valid, i.e., they do not violate any constraint in C.

After having verified that a test suite is valid, it is important to further investigate about its completeness, meaning that all feasible tuples of values for parameters must be covered. Formally:

Definition 3 (Complete Test Suite)**.** Let $M = (P, D, C)$ be a constrained combinatorial model. Given a combinatorial test suite TS and being t the strength for test generation, we say that TS is *complete* if any valid duple tp of size t is covered by at least a test in TS.

```
Model MVM

Parameters:

 State : {OFF, STARTUP, SELF_TEST, VENTILATION_OFF, PCV_INSP, PCV_EXP, PSV_INSP, PSV_EXP}
 NextState : {OFF, STARTUP, SELF_TEST, VENTILATION_OFF, PCV_INSP, PCV_EXP, PSV_INSP, PSV_EXP}
 InValve : {OPEN CLOSE}
 OutValve : {OPEN CLOSE}
 Mode: {PCV PSV}
 PowerOff : Boolean
 SelfTestPassed: Boolean
 StartupEnded: Boolean
 StartVentilation: Boolean
 InspTimePassed: Boolean
 ExpTimePassed: Boolean
 StopVentilation: Boolean
 Resume: Boolean

Constraints:

 # NextState = OFF <=> PowerOff #
 # NextState = SELF_TEST <=> (State = STARTUP AND StartupEnded)#
 # SelfTestPassed <=> State = SELF_TEST #
 # Resume <=> State = SELF_TEST #
 # StartupEnded <=> State = STARTUP #
 # (State = SELF_TEST AND (SelfTestPassed OR Resume)) => NextState = VENTILATION_OFF #
 # (InValve = OPEN AND OutValve = CLOSE) OR (InValve = CLOSE AND OutValve = OPEN) #
 # (InValve = OPEN <=> (State = PSV_INSP OR State = PCV_INSP)) AND (OutValve = OPEN <=> (State = PSV_EXP OR State
       = PCV_EXP)) #
 # (State = VENTILATION_OFF AND Mode=PCV AND StartVentilation) => NextState = PCV_INSP #
 # (State = VENTILATION_OFF AND Mode=PSV AND StartVentilation) => NextState = PSV_INSP #
 # (State = PSV_INSP AND Mode=PSV AND InspTimePassed) => NextState = PSV_EXP #
 # (State = PCV_INSP AND Mode=PCV AND InspTimePassed) => NextState = PCV_EXP #
 # (State = PCV_EXP AND Mode=PCV AND ExpTimePassed) => NextState = PCV_INSP #
 # (State = PSV_EXP AND Mode=PSV AND ExpTimePassed) => NextState = PSV_INSP #
 # (State = PSV_INSP AND Mode=PCV AND InspTimePassed) => NextState = PCV_EXP #
 # (State = PCV_INSP AND Mode=PSV AND InspTimePassed) => NextState = PSV_EXP #
 # (State = PCV_EXP AND Mode=PSV AND ExpTimePassed) => NextState = PSV_INSP #
 # (State = PSV_EXP AND Mode=PCV AND ExpTimePassed) => NextState = PCV_INSP #
 # (StopVentilation => (State = PCV_EXP OR State = PSV_EXP)) => NextState = VENTILATION_OFF #
 # (InspTimePassed AND NOT ExpTimePassed) OR (ExpTimePassed AND NOT InspTimePassed) #
 # PowerOff <=> (NOT SelfTestPassed AND NOT StartupEnded AND NOT StartVentilation AND NOT InspTimePassed AND
       NOT ExpTimePassed AND NOT StopVentilation AND NOT Resume) #
 # SelfTestPassed <=> (NOT PowerOff AND NOT StartupEnded AND NOT StartVentilation AND NOT InspTimePassed AND
       NOT ExpTimePassed AND NOT StopVentilation AND NOT Resume) #
 # StartupEnded <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartVentilation AND NOT InspTimePassed AND
       NOT ExpTimePassed AND NOT StopVentilation AND NOT Resume) #
 # StartVentilation <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartupEnded AND NOT InspTimePassed AND
       NOT ExpTimePassed AND NOT StopVentilation AND NOT Resume) #
 # InspTimePassed <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartupEnded AND NOT StartVentilation AND
       NOT ExpTimePassed AND NOT StopVentilation AND NOT Resume) #
 # ExpTimePassed <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartupEnded AND NOT StartVentilation AND
       NOT InspTimePassed AND NOT StopVentilation AND NOT Resume) #
 # StopVentilation <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartupEnded AND NOT StartVentilation AND NOT
       InspTimePassed AND NOT ExpTimePassed AND NOT Resume) #
 # Resume <=> (NOT PowerOff AND NOT SelfTestPassed AND NOT StartupEnded AND NOT StartVentilation AND NOT
       InspTimePassed AND NOT ExpTimePassed AND NOT StopVentilation) #
```

Listing 6.1: Example of a combinatorial model for the MVM case study.

6.2 Combinatorial sequence testing

Combinatorial testing is generally applied by taking into account the inputs of a system. However, for event-driven software, the inputs to be considered are *events*. Moreover, in such systems, there may be constraints on the events that can be used as input during testing. For example, a SUT may require that a given event `read` appears after another event `open`, and if a test does not meet this constraint, the test is invalid and cannot be applied (see Def. 2). These constraints are based on temporal precedences and are not normally supported by regular combinatorial test generators.
In this section, I present a method that can be applied to generate test sequences with combinatorial coverage for Finite State Machines (FSMs), which are commonly used for representing event-driven systems, such as the PHD protocol (previously discussed in Sect. 5.2.2) [172]. This approach is known as *combinatorial sequence testing* (CST) [113] and supports FSMs in the form of Mealy machines, which are a rather general implementation of FSMs.

6.2.1 Finite State Machines

Finite State Machines (FSMs) are commonly used for modeling event-driven software. In particular, the most used ones are *Mealy* machines, since they allow one not only to manage states and input events, but also output events.

Definition 4 (Mealy machine)**.** A *Mealy machine F* is a 6-tuple $(S, s_0, \Sigma, \Lambda, T, G)$ in which:

- S is a finite set of states;
- $s_0 \in S$ is the initial state of the machine F;
- Σ is a finite set that represents the input alphabet;
- Λ is a finite set representing the output alphabet;
- $T : S \times \Sigma \rightarrow S$ is the transition function that maps pairs of a state and an input symbol to the corresponding next state;
- $G : S \times \Sigma \rightarrow \Lambda$ is the output function that maps pairs of a state and an input symbol to the corresponding output symbol.

In practice, FSMs and Mealy automata may not be complete. In fact, certain inputs may not be defined in some states. For this reason, the definition of FSM can be extended by adding the notion of *incompleteness*.

Definition 5 (Complete and incomplete FSM)**.** Given an FSM $F(S, s_0, \Sigma, \Lambda, T, G)$ we say that F is a *complete machine* iff for all $s \in S$ and for all $e \in \Sigma$ the transition function $T(s, e)$ and the output function $G(s, e)$ are defined. On the contrary, we say that F is a *incomplete machine* iff there exist

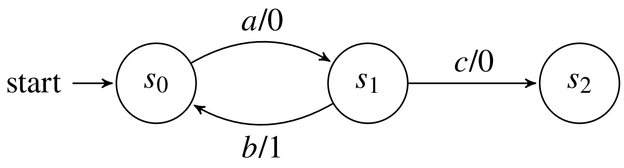


Figure 31: Example of Mealy machine. On the arrows, the pair $input/output$ **is reported.**

a state $s \in S$ and an event $e \in \Sigma$ for which the transition function $T(s, e)$ is not defined (neither is $G(s, e)$).

For example, Fig. 31 represents an incomplete machine, because the transition function T is not defined for the input symbols b and c in the state s_0, for the input symbol a in the state s_1, and for the symbols a, b, and c in the state s_2.

6.2.2 Combinatorial sequence testing of FSMs

In classical combinatorial testing, testers are interested in covering the interaction among a fixed set of inputs, each with a given set of possible values. Instead, in combinatorial sequence testing (CST) [113] of FSMs, it is necessary to focus on covering the interaction of inputs taken from a unique set (the input alphabet) but provided to the system under test in different orders.
This difference requires the redefinition of "test" as a sequence of inputs of variable length. In the proposed approach, a test is a finite sequence of events $ts = (e_1, e_2, \ldots, e_n)$ all belonging to the input alphabet Σ. Then, a test suite composed of all tests ts_i is created so that the desired combinatorial coverage of the input combinations is reached.

Definition 6 (Combinatorial sequence coverage)**.** We say that a test suite achieves the t-way combinatorial sequence coverage iff for any tuple of t inputs there exists a test sequence in which these t inputs occur in any possible order (allowing interleaving extra inputs among the elements of the tuple).

As previously explained, most event-driven software can be represented using incomplete FSMs since, in some states, some events cannot be fired. This assumption implicitly defines some *constraints* on the FSM, meaning that only some test sequences are valid, while others are not. For this reason, as for a single test case (see Def. 2), the validity of a test sequence has to be defined.

Definition 7 (Valid test sequence)**.** Given an FSM $F(S, S_0, \Sigma, \Lambda, T, G)$ as per Def. 4, let $ts = (e_1, e_2, \ldots, e_n)$ be a test sequence composed of a sequence of n events. Assume that ts^i is the list of events in ts starting from e_1 to e_i and $s(ts^i)$ is the state reached starting from the initial state s_0 applying all events in ts^i. We call ts a *valid test sequence* if and only if, for all $e_i \in ts$, e_i can be fired starting from the state $s(ts^{i-1})$, i.e., $T(s(ts^{i-1}), e_i)$ and $G(s(ts^{i-1}), e_i)$ are both defined.

When it comes to the generation of test sequences, the most common approach is the one that extends classical combinatorial testing algorithms in order to generate SCAs [113]. However, the main limitation of this approach is that all sequences must be composed of the same number of steps. Instead, in the automata-based approach presented in this book, it is possible to have tests with different lengths. This is a more realistic way to test systems, since not all user interactions are equal or of the same duration.
Considering that every test is a permutation of events, first, I introduce the concept of automata representing a t-wise permutation of t events.

Definition 8 (T-wise automaton)**.** Given a permutation p of t events $(e_1, e_2, \ldots, e_t)$, the automaton $\mathcal{A}$ built as in Fig. 32 is called the t-wise automaton. We call `automaton(p)` the function that builds the t-wise automaton that represents the tuple p.

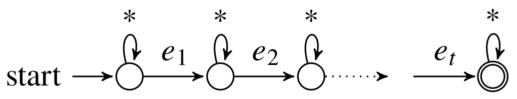


Figure 32: Example of an automaton representing the sequence $(e_1, e_2, \ldots, e_t)$**.**

Note that a t-wise automaton $\mathcal{A}$ can be used to check whether a sequence ts covers the tuple represented by $\mathcal{A}$: if ts is accepted by $\mathcal{A}$, then the tuple is covered; otherwise, it is not.

6.2.3 Algorithm for CST

The automata-based algorithm for CTS is based on the concept of T-wise automaton. In fact, it is possible to perform logical and mathematical operations between automata: if we consider the automaton $\mathcal{A}_1$ covering the tuple tp_1 and the automaton $\mathcal{A}_2$ covering the tuple tp_2, their intersection $\mathcal{A}_1 \cap \mathcal{A}_2$ (if not empty) covers both tp_1 and tp_2.
In this way, the CST problem is solved by Algorithm 1. It implements a one-test-at-the-time test generator. First, the algorithm builds an empty automaton A, then tries to randomly add as many t-wise automata as possible. This process is commonly known in the literature as *collecting*: multiple t-wise automata are collected in a unique automaton. When the automaton cannot be modified by adding any other t-wise automaton, any string that can be derived from A represents a test that covers all the permutations from which the t-wise automaton is built. In Algorithm 1, this operation is performed by the function `string(A)` which returns the shortest string accepted by the automaton A.
Note that since some users may prefer to obtain shorter test sequences and others may prefer longer ones, when setting the algorithm, the user is required to set the maximum number N of automata to be collected together.

Algorithm 1 Algorithm for test generation.

```
Require: $I$ the set of events
Require: $t$ the strength of the tests
Require: $N$ the max number of tuples for each test sequence
Ensure: $TS$ the test suite for CST
  $T \leftarrow$ t-permutations of $I$
  $TS \leftarrow \emptyset$
  $i \leftarrow 0$
  $A \leftarrow$ empty automaton
  while $T \neq \emptyset$ do
      $p \leftarrow$ a random element in $T$
      $a \leftarrow automaton(p)$
      if $a \cap A \neq \emptyset$ then
          $A \leftarrow a \cap A$
          $T \leftarrow T - \{p\}$
          $i \leftarrow i + 1$
          if $i \geq N$ then
              AddTest($TS$,$A$)
              $i \leftarrow 0$
          end if
      end if
  end while
  AddTest($TS$,$A$)

  procedure AddTest($TS$,$A$)
      $TS \leftarrow TS + string(A)$
      $A \leftarrow$ empty automaton
  end procedure
```

However, this standard algorithm may generate invalid test sequences, as no information about the system and its constraints is considered. Therefore, in the following section, three different approaches to repair invalid tests are analyzed:

- **Reject_not_valid (REJ)**: if a sequence contains an event that is invalid at the time it is applied, the whole sequence is rejected.
- **Stop_at_error (STP)**: if a sequence contains an event that is invalid at the time it is applied, the sequence is executed only until the error is reached. The following events are not tested.
- **Skip_error (SKP)**: if a sequence contains an event that is invalid at the time it is applied, the single event is skipped and the following events are executed.

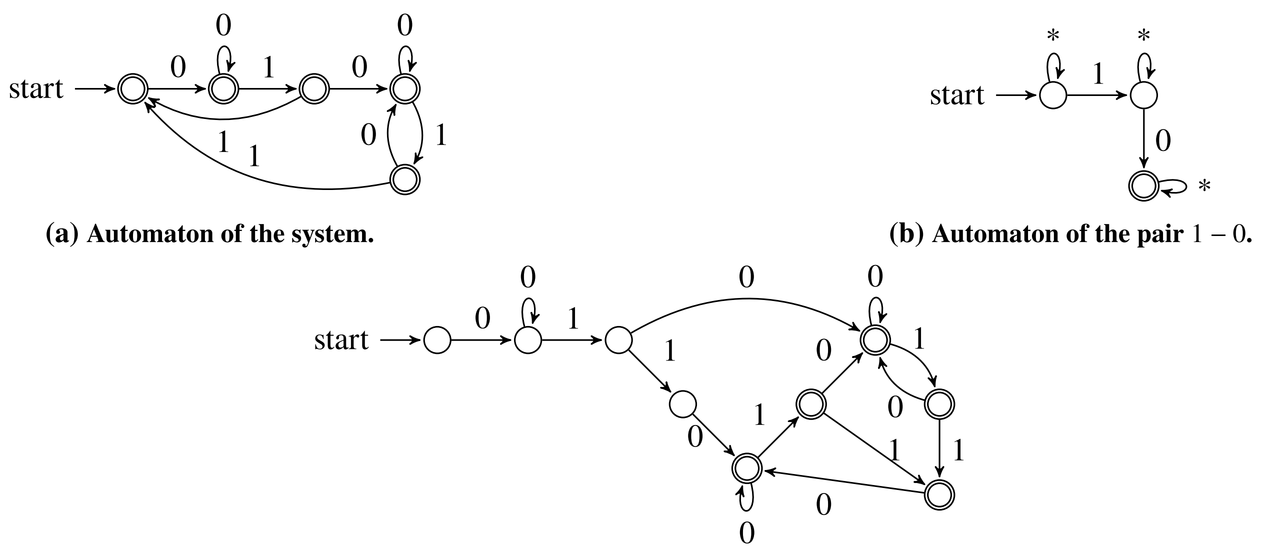
Algorithm 2 Algorithm for test generation.

```
Require: $I$ the set of events
Require: $F$ the finite state machine
Require: $t$ the strength of the tests
Require: $N$ the max number of tuples for each test sequence
Ensure: $TS$ the test suite for CST
 1: $T \leftarrow$ t-permutations of $I$
 2: $TS \leftarrow \emptyset$
 3: $i \leftarrow 0$
 4: $A \leftarrow automaton(F)$                                ▷ init A with the FSM automaton
 5: while $T \neq \emptyset$ do
 6:     $p \leftarrow$ a random element in $T$
 7:     $a \leftarrow automaton(p)$
 8:     if $a \cap A \neq \emptyset$ then
 9:         $A \leftarrow a \cap A$
10:         $T \leftarrow T - \{p\}$
11:         $i \leftarrow i + 1$
12:         if $i \geq N$ then
13:             AddTest($TS$,$A$)
14:             $i \leftarrow 0$
15:         end if
16:     else if $automaton(F) \cap a = \emptyset$ then                ▷ $p$ is infeasible
17:         $T \leftarrow T - \{p\}$
18:     end if
19: end while
20: AddTest($TS$,$A$)
21:
22: procedure AddTest($TS$,$A$)
23:     $TS \leftarrow TS + string(A)$
24:     $A \leftarrow automaton(F)$                            ▷ init A with the FSM automaton
25: end procedure
```

How to generate only valid test sequences

In practice, one may want to generate from the beginning only valid tests without the need of repairing them. For this reason, Algorithm 1 can be modified as reported in Algorithm 2. In this new version of the generation algorithm, called CNST, t-wise automata are collected not starting from an empty automaton, but from the one that accepts only valid sequences of inputs for the FSM under test (lines 4 and 24). The operation is exemplified in Fig. 33.

If the FSM of the system under test does not accept a given tuple, this means that the tuple is *infeasible* (it clashes with the constraints of the system). This screening is done by the algorithm at line 16: it checks if a tuple tp that cannot be collected with the current automaton can instead be collected with the automaton containing only the constraints of the FSM ($automaton(F)$). If not, the tuple is infeasible.



(a) Automaton of the system.

(b) Automaton of the pair $1 - 0$.

(c) Intersection among the two previous automata.

Figure 33: Intersection process among automata for the pattern recognition system.

Algorithm 3 Monitoring.

```
 1: procedure AddTest($TS$,$A$,$T$)
 2:     $test \leftarrow string(A)$
 3:     for all $t \in T$ do
 4:         if $isAccepted(test, automaton(t))$ then
 5:             $T \leftarrow T - \{t\}$
 6:         end if
 7:     end for
 8:     $TS \leftarrow TS + test$
 9:     $A \leftarrow automaton(F)$
10: end procedure
```

Monitoring

To further optimize test generation, a commonly adopted technique is *monitoring* (previously introduced in Sect. 5.4.1). It consists in checking if a test generated for a set of tuples accidentally covers other tuples as well. Algorithm 3 shows how monitoring works: once a test is generated, all tuples that are not yet covered are checked against the new test. If a new tuple is accidentally covered, it is discarded. Note that, as previously done for the t-wise automaton, to check if a tuple is covered by a test, it is possible to verify whether the automaton representing that tuple accepts the test sequence. One may argue that applying monitoring could slow down the process. However, while the collecting procedure shown in Algorithm 2 can be expensive, since it requires the operation of intersection among automata, the monitoring is generally much faster, since acceptance is easily computed.

# Automata per test sequence (N)	10
# Transitions	65
# States	5
# Events	23
# Event pairs	529
# Valid event pairs	484
# Event triples	12,167
# Valid event triples	10,648

Table 19: PHD benchmark characteristics.

Monitoring	**Method**	**# Seq.**	**Max. Len.**	**Min. Len.**	**Avg. Len.**	**Tot. Len.**	**# Valid Seq.**	**# Cov. pairs**	**# Cov. states**	**# Cov. trans.**	**Gen. t [s]**
NO	CNST	41	20	11	17	708	41	484	5	51	428.40
NO	SKP	45	20	2	15	693	0	270	5	39	135.60
NO	REJ	45	19	2	15	701	0	0	0	0	144.24
NO	STP	45	20	2	15	692	0	49	2	12	150.80
YES	CNST	41	21	15	17	723	41	484	5	51	474.92
YES	SKP	45	20	2	15	686	0	271	5	39	185.08
YES	REJ	45	18	2	15	695	1	1	1	2	131.00
YES	STP	45	18	2	15	701	0	55	3	16	168.05

Table 20: Method evaluation (pairwise testing).

6.2.4 Method evaluation

In this section, the proposed approach is evaluated and applied to a real medical case study: the PHD protocol (see Sect. 5.2.2). More details about the characteristics of the PHD protocol and on how the CST algorithm has been configured are reported in Tab. 19. Furthermore, the evaluation has been carried out by performing 10 times each experiment, in order to reduce the possible non-determinism. The average results, obtained on a computer with 14 GB of RAM and an Intel® Core™ i5-750 CPU, are reported in Tab. 20. The code generating the test sequences is available at `https://github.com/fmselab/FiniteStateMachineCombinatorial`, and has been implemented using the `dk.brics.automaton` [136] Java library.

In the following, several aspects about the impact of the proposed approach and its configuration parameters are discussed in detail.

Method	% Valid Seq.	% Pairs Cov.	% States Cov.	% Transitions Cov.
CNST	100.00	100.00	100.00	77.65
SKP	3.92	55.50	100.00	62.94
REJ	2.94	0.86	27.50	5.88
STP	1.96	12.52	75.00	28.24

Table 21: Evaluation of the results obtained with different generation methods for the PHD case study.

Sequence generation time with the CNST method

Observing the *generation time*, it is evident that the CNST method is the slowest, as the generation of the sequences in accordance with the constraints of the FSM requires more time than repairing the sequences. In fact, building the intersection among automata is time-consuming since they must contain the entire system constraints from the beginning (instead of starting from the empty automaton). However, CNST leads to better results in terms of coverage, as discussed in the following.

Coverage and valid sequences with CNST method

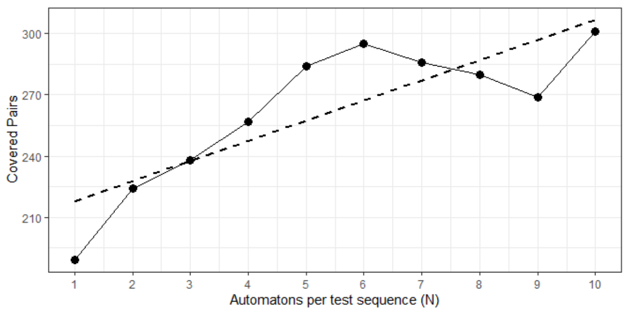
Table 21 reports the result of the coverage reached in the case of the PHD protocol with the sequences generated by all the methods presented before. The results obtained confirm that the CNST method, which generates test sequences following the constraints imposed by the FSM of the system, leads to better (or equal) results than the other reparation approaches:

- The percentage of valid sequences is higher. With other methods, in many cases, no valid sequences are produced. In those cases, the user must repair the sequences (with one of the three proposed approaches - SKP, REJ, or STP) to still perform the testing activity;
- The overall coverage (event pairs, states, and transitions) is higher or the same for CNST compared to the one obtained with other methods because all sequences that are generated by the algorithm can be executed on the system, as they contain only valid events.

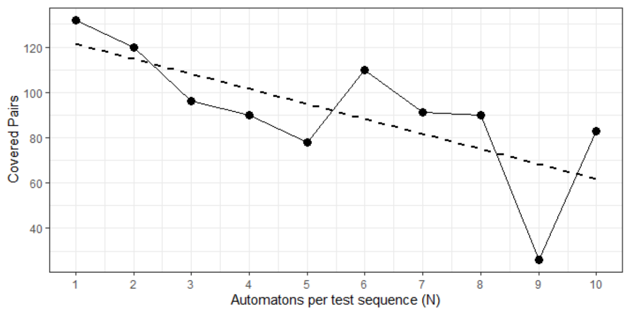
Note that the value of covered pairs is computed only over the number of feasible ones, since some of them may not be possible to be covered due to the constraints of the system.

Impact of monitoring

Observing the results in Tab. 20, it can be seen that the test suites obtained when using monitoring and methods that involve the repair of the sequences generally have better or equal coverage (number of pairs covered) than those that do not use monitoring. This is reasonable because, without monitoring,



(a) With the SKP method.



(b) With the STP method.

Figure 34: Number of pairs covered with different values of the parameter *N*.

the algorithm produces more sequences that can fail, and, in some cases, when the test sequence is invalid, its execution must be halted before its termination.

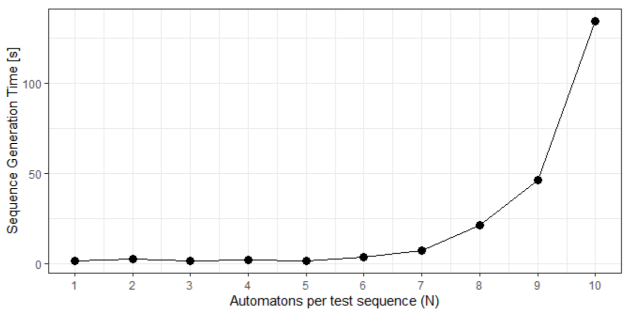
Furthermore, in these experiments, the monitoring optimization has shown not to be time consuming, as only a little overhead is added in some cases. For this reason, it can be concluded that the application of monitoring is always a good choice.

Correlation between the number of covered pairs and N

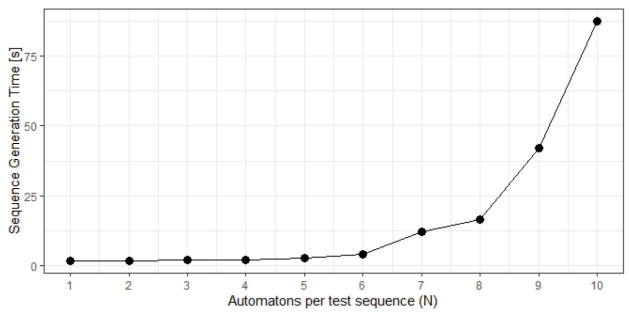
When test sequences need to be repaired, the number of pairs covered by the generated test suite is influenced by the value chosen for the parameter N (number of automata per batch). In fact, as shown in Fig. 34a, for the SKP repair method, the number of pairs covered has a growing trend with increasing N. This is justified by the fact that, when long sequences are generated, more pairs are included in each. Therefore, considering that the events that cannot be covered are skipped, more pairs are covered by fewer test sequences. On the other hand, Fig. 34b shows that for the STP repair method, the number of covered pairs has a decreasing trend with the growth of N: having longer sequences means that, when an invalid event is reached, the execution of the whole sequence is stopped and more events are not executed. A similar behavior can be observed when using REJ. In contrast, if the CNST method is chosen, the number of pairs covered remains constant when N varies, since all pairs in a test sequence always satisfy all constraints. In conclusion, for the SKP method, a higher N can improve coverage, while for STP and REJ, the shorter the better.

Correlation between generation time and N

Testers may arbitrarily choose the value of N (number of automata to be collected in each test sequence) depending on the generation and repairment method chosen, but experiments show that the sequence generation time increases exponentially with the increment of N. In fact, the intersection



(a) With the STP method.



(b) With the CNST method.

Figure 35: Sequence generation time $[s]$ with different values for the parameter N.

of N automata usually requires much more time than the intersection of $N - 1$ automata, especially when higher values of N are used.

This behavior is reflected by Fig. 35a that shows the correlation between generation time and N when test sequences are generated without taking into account the constraints (in the example, the STP method is used). Furthermore, even when the CNST method is used, experiments show that the correlation between N and the generation time is exponential (see Fig. 35b). However, increasing N leads to smaller test suites, as shown below, so users should decide the value of N taking into account this trade-off.

Comparison between automata-based generation method and SCAs

Common approaches for CST are based on *sequence covering arrays* (SCAs). In order to validate the effectiveness of the proposed method, in the following, the proposed approach is compared with the one using SCAs (using the tool provided by [141]). To gather more results and investigate the applicability with a strength greater than $t = 2$, the tests used in this section to evaluate the automata-based approach are obtained using 3-wise testing (see Tab. 22).

SCAs generators only perform permutations of n events, thus they do not consider the constraints of the system. For this reason, in order to compare the results of the approach proposed with the one generating SCAs, test sequences need to be repaired. Moreover, since the standard approach with SCAs produces sequences all of the same length (equal to the number of events considered), test sequences are shorter than those obtained with the automata-based method presented in this section. These two aspects combine and, as a consequence, the generation time is shorter when using SCA: computing all the permutations is less complex than computing the intersection among automata, especially if it is performed without considering the constraints. Also, repairing test sequences does not increase the generation time as CNST does.

Monitor.	**Method**	N	**# Seq.**	**Max. Len.**	**Min. Len.**	**Avg. Len.**	**Tot. Len.**	**# Valid Seq.**	**# Cov. triads**	**# Cov. states**	**# Cov. trans.**	**Gen. t [s]**
NO	CNST	6	1,775	22	10	15	28,076	1,775	10,648	5	65	538.37
NO	SKP	6	1,521	22	12	16	25,198	0	7,075	5	65	981.22
NO	REJ	6	1,521	21	12	16	25,178	0	0	0	0	827.73
NO	STP	6	1,521	22	12	16	25,277	0	1,298	5	42	881.79
YES	CNST	6	1,775	23	11	15	28,032	1,775	10,648	5	65	7475.46
YES	SKP	6	1,521	21	12	16	25,159	0	7170	5	65	7754.58
YES	REJ	6	1,521	21	12	16	25,165	0	0	0	0	7329.89
YES	STP	6	1,521	21	11	16	25,194	0	1,263	5	39	7109.64

Table 22: Method evaluation (3-wise testing).

	Method	**Valid Seq.**	**Triads Cov.**	**States Cov.**	**Trans. Cov.**	**Avg. Cov**
Automata-based	CNST	100.00%	100.00%	100.00%	95.29%	98.43%
Automata-based	SKP	0.06%	66.77%	100.00%	95.29%	
	REJ	0.06%	0.07%	25.00%	5.88%	51.10%
	STP	0.06%	12.00%	92.50%	62.35%	
SCAs	SKP	2.17%	26.44%	85.00%	57.65%	
	REJ	2.17%	0.00%	15.00%	2.35%	29.03%
	STP	2.17%	0.15%	50.00%	24.17%	

Table 23: Comparison between SCAs and automata-based method (3-wise testing).

Tab. 23 shows the summary of the comparison between the coverage obtained by the automata-based method and the one by SCAs (with different repairing procedures). The results confirm that the former performs better than the latter in every aspect analyzed (except for the time). The only aspect in which the SCA-based method outperforms the automata-based one is the percentage of valid sequences (when CNST is not used). However, this is not an unexpected result since SCAs generate fewer test sequences. All the results obtained show that the automata-based method is overall better than the SCAs one, and it is confirmed by the *paired t-test* [178] with:

- H_0: the two methods perform in the same way, in terms of coverage
- $\alpha = 0.05$
- $t = 3.8507$
- $p_{value} = 0.004873$

The statistical test ends with $p_{value} < \alpha$, so the null hypothesis can be rejected and, since the test value $t > 0$ it is confirmed that the automata-based method performs better than SCAs. This is an important result, especially when CST is applied to safety-critical devices (such as in the medical domain), as higher coverage can lead to higher-quality products and to a greater number of possible bugs revealed.

6.3 Comparing combinatorial test generators

Since combinatorial testing has proven to be very effective in a variety of systems, new combinatorial test generators are proposed every year. These tools exploit different algorithms to generate tests. However, if a limited amount of resources is available, testers may need a measure to define which tool should be preferred for generating tests. Benchmarking combinatorial test generators in a fair and effective way is a difficult task, and there is no well-established methodology or environment for generator comparison yet. In the following, I present some complexity measures for comparing combinatorial models, a formula for computing the cost of using a specific test generator, and a benchmarking environment that can be used during the preliminary phases of testing in order to evaluate the impact of tools on the cost of testing and to guide the choice on the most suitable one.

6.3.1 Model complexity measures

Each combinatorial model may have different complexity due to multiple parameters and constraints. Several complexity measures have been proposed in the literature. Among them, the most used are the following:

- *Number of parameters*: having more parameters means having more combinations to check.
- *Size*: it represents the total number of distinct tests (valid and invalid) that can be generated, corresponding to the product of the cardinalities of all domains. Having more possible combinations involves a higher complexity in terms of execution time.
- *Number of constraints*: more constrained models may need more constraint checks, which complicate the test generation procedure.
- *Number of logical operators in the constraints*: CIT models may have many simple constraints or a few constraints that are composed of several logical operators. Thus, this definition of model complexity is not limited only to the number of constraints but considers their complexity as well, computed as the sum of the number of logical operators.

Besides the measures given above, two more measures that refer only to the *semantics* of the models can be used:

- *Tuple Validity Ratio*: given a strength t, it represents the fraction of valid t-tuples over the total number of t-tuples. One way to compute the ratio is to enumerate all the tuples and check if they are valid or not. To check the validity of a tuple, any constraint or logical solver may be used.

- *Test Validity Ratio*: it is intended as the fraction of valid tests over the total number of possible tests. It can be computed by enumerating all tests and checking whether they are valid or not. In practice, this may require an exponential time with the size of the problem, and it is not doable for large models. However, the literature reports a better way that does not require the enumeration of the tests and is based on the use of Multi-valued Decision Diagrams [91] that can compute the number of valid models in a very efficient way.

6.3.2 Computing the cost of test generators

In practice, it is not trivial to identify the best generator. In fact, since test generation for CIT is a multi-objective problem, it must take into account both time and test suite size. Indeed, sometimes, generators can generate a lot of tests in less time or generate a small but still complete test suite in hours of computation.

For this reason, a formula for cost estimation must consider three different parameters, as proposed by the cost model in [93]:

$$cost = time_{total} = time_{gen} + size \cdot time_{test} \tag{6.1}$$

The *cost* is equal to the total testing time, composed by the $time_{gen}$ used by the generators to generate test suites and the product between the $size$ of the test suite and the average $time_{test}$ required to perform a single test on the SUT. This cost model can be used as a method to evaluate a tool with respect to another.

Note that testers may need to consider other aspects beside the cost. For example, some tools may not be able to manage a specific constrained combinatorial model (e.g., because it uses expressions not supported by the generator, such as relational expressions). In this case, only the generators capable of handling the model must be considered, and, between them, the least expensive one should be chosen by the testers.

Feature:		# models
Parameters		
	all with the same cardinality	5
	only booleans	4
	with also enumeratives	18
	with also integers	33
Constraints		
	without constraints	22
	as forbidden tuples	33
	in Clausal Normal Form	25
	containing relational operators (>, <, etc.)	6

Table 24: Summary of the benchmarks features.

6.3.3 A benchmarking environment for CIT tools

To allow testers to evaluate combinatorial test generators, the CTWedge environment [87] has been extended and refactored, to facilitate the addition of new generator tools. In this way, users can test the available tools against several benchmark examples and define which is the best for the application scenario.

Benchmarks

In the benchmarking environment integrated into CTWedge, 196 test models have been collected. Some benchmark models were already embedded in CTWedge as example models (previously taken from [93,107,146,147,160]), others have been collected from the PICT GitHub page [1], and others have been extracted from the collection used in [167]. Note that the scenarios are not referred specifically to the medical software domain, but their complexity is distributed in a way that conclusions drown from them are applicable to every application domain.

In order to obtain a fair comparison, all the gathered models that were not written in the CTWedge language have been translated, both in an automatic and manual manner. Benchmark models vary greatly in terms of number of variables, number of constraints, total size, and tuple and test validity ratios. Moreover, even in terms of relevant features (see Tab. 24), models differ significantly. Note that a model can exhibit more than one feature, while others none of them. By using these benchmarks, and possibly additional ones, test teams may try the available generators on models that are more similar to the one of the system to be tested and extract measures useful for defining which tool is the best fit for the specific application scenario.

[1]`https://github.com/microsoft/pict`

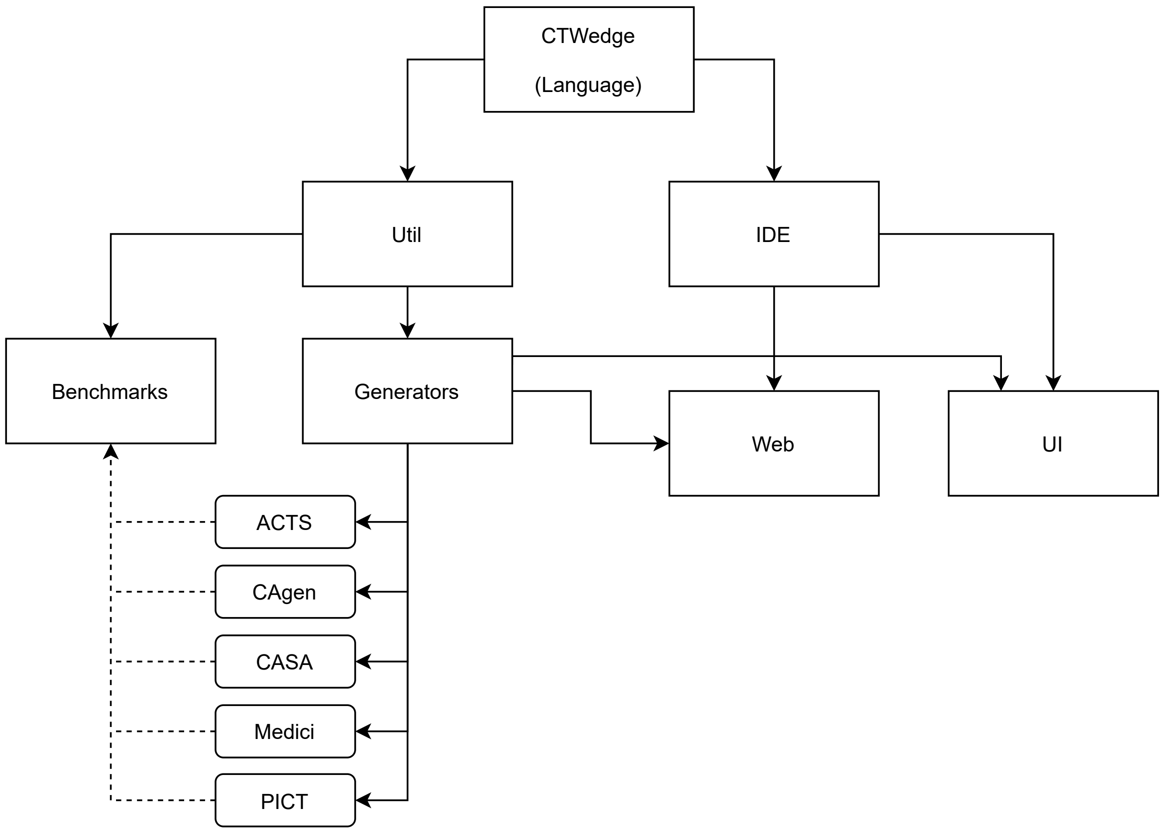


Figure 36: Refactored architecture of CTWedge for benchmarking integration.

How to integrate new CIT generators

To make the integration of new generators into the benchmarking environment easier, the internal architecture of CTWedge has been deeply refactored (Fig. 36), by exploiting Eclipse extension points, made available by the Eclipse plugin development environment. The benchmarking framework already integrates the following CIT generators:

- **ACTS** (Automated Combinatorial Testing for Software), developed by NIST [181].
- **CASA** (Covering Arrays by Simulated Annealing), which generates combinatorial test suites using simulated annealing [92];
- **CAGen** (Covering Array Generation), developed by SBAresearch [173];
- **Medici** (Multi-valuEd Decision diagrams for Combinatorial Interaction Testing), developed by University of Bergamo [91];
- **PICT** (Pairwise Independent Combinatorial Testing), developed by Microsoft. [134].

However, additional generators may be added using the procedure explained in the following.

Each generator must extend the class `ICTWedgeTestGenerator`, implement the method `getTest-Suite(...)`, and extend the `ctwedge.util.ctwedgeGenerators` extension point[2]. The `getTest-`

[2]More detailed information about how to integrate new generators are available at `https://github.com/fmselab/ctwedge/wiki`

`Suite(...)` method must return the generated test suite for a defined model, together with the required generation time.

This way of extending the benchmarking environment allows for a loosely coupled structure: the benchmarking environment does not need modifications when new generators are added, since they are automatically discovered by Eclipse as new plugins. For this reason, each generator must be implemented as an Eclipse plugin managed with Maven. As an additional requirement, each generator needs to also include a translator from CTWedge grammar to its own one or, at least, shall support CTWedge as input format.

After having integrated a generator (or chosen an already available one), the benchmark can be executed by the functionalities in the package `ctwedge.generator.benchmarks`, and, in particular, in the class `BenchmarkTest`. This class exploits the *Eclipse extension point* extended by all generators (i.e., `ctwedge.util.ctwedgeGenerators`) to discover all generators that have been defined in the environment and check the test suite sizes and generation times for each generator on each benchmark model.

Validation and test suite checking activities

The proposed benchmarking framework provides completeness and validity checks, suitable for verifying if a tool generates test suites that violate the properties described in Sect. 6.1. These APIs are contained in the package `ctwedge.util.validation` and, in particular in the `SMTTestSuiteValidator` class that implements the validation functionalities by exploiting an SMT solver. Given a test suite `TS`, the main functionalities offered by the validation APIs are as follows:

- `isValid()`, returning whether the test suite satisfies all the constraints contained in the CIT model;
- `howManyTestsAreValid()`, which counts how many tests contained in the test suite are valid;
- `isComplete()`, which checks the completeness of the test suite by verifying if some valid combination of parameters is not covered by the test suite;
- `howManyTuplesCovers()`, returning the number of tuples covered by all the tests in the test suite.

6.4 Parallelizing combinatorial test generation

Taking into account the cost function for the generation of combinatorial test suites (Formula 6.1), the cost of choosing a specific tool can be reduced if the test suite $size$ or generation $time_{gen}$ is reduced.

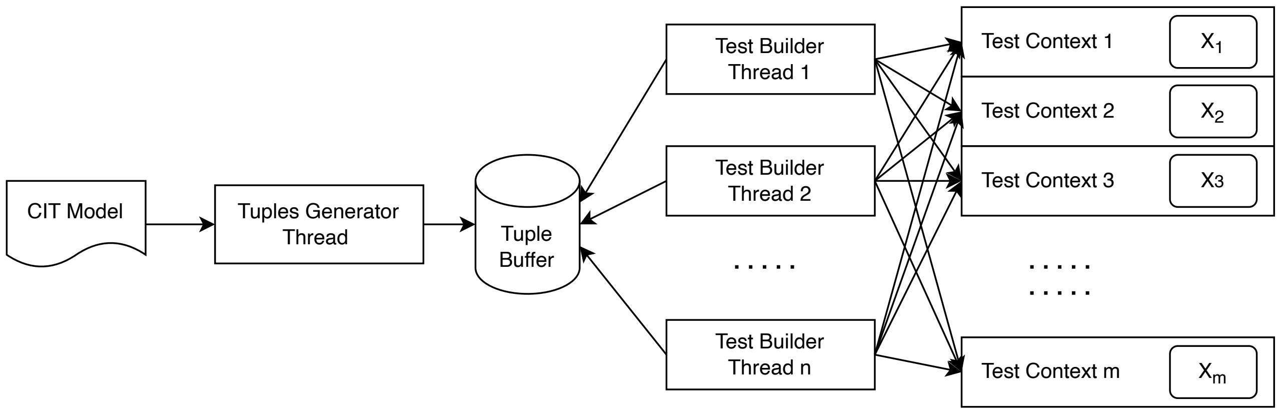


Figure 37: Structure of the tools for multi-thread combinatorial test generation.

In fact, the time for executing a single test, $time_{test}$, depends on the system to be tested, and it can not be modified by varying the test generator.

Although reducing test suite size may be difficult (since it requires the implementation of new algorithms, which have to be more effective than IPO-G [117], that has proved to be the cutting-edge tool), reducing the generation time is easier. For this reason, in this section, I present two similar approaches that exploit multi-thread, different underlying representations, and aim to speed up the test generation process. Both approaches are based on the structure shown in Fig. 37 and explained in general in the following. Then, a specific description of the process, which depends on the internal implementation, is given in Sect. 6.4.1 and Sect. 6.4.2.

In the multi-threaded approach, all the tuples to be covered are generated, exploiting the Cartesian product, by a single thread from a CIT model (such as the one of the MVM case study in Listing 6.1) and stored in a shared buffer with limited capacity (40 tuples in our experiments, but it can be configured by the user). Then, when the shared buffer is fully filled, the tuple generation thread stops and waits until a new free slot is available. This implementation allows avoiding storing all tuples at the same time, and thus guarantees a consistent saving in memory utilization, especially for complex combinatorial models, in which the number of tuples can be significantly high.

When at least a tuple is available in the tuple buffer, *n Test Builder* threads (where *n* can be automatically selected by the tool depending on the hardware architecture or specified by the user) start to consume each tuple tp_i. The tuples are consumed exploiting several *Test Contexts*, which are continuously updated as each tuple is managed by a single *Test Builder* thread.

Definition 9 (Test context)**.** Let $P = \{I, C\}$ be a combinatorial problem, where $I = \{x_i, v_i\}$ is a set of parameters (each x_i with the set of possible values v_i) and C is the set of constraints. We call $TC = \langle A, X_{TS} \rangle$ a *test context* for a combinatorial problem P, where A is a list of assignments of some parameters p_i to one of their possible values $v_{i,j}$ and X_{TS} is the structure used for the management of the combinatorial problem P, its constraints, and the assignments committed to the context so far.

As reported in Def. 9, each test context TC is embedded with a list of assignments A, that represents a partial test case T, together with the structure containing all the information about the model (parameters and constraints) and the test itself. The process that updates a test context is repeated until it is *complete*, i.e., it represents a complete test case, that is obtained when the list of assignments A includes all the parameters of the model. Given a test context TC, a tuple tp can be:

- *implied*, if all assignments of the tuple tp are already contained in the assignments A, i.e., the possible partial test case T;
- *compatible*, if the tuple tp contains only assignments that are not in conflict with those of the test context TC, i.e., in the test TC, each parameter contained in the tuple tp is still not valorized or has an equal value, and tp does not clash with the constraints of the combinatorial problem;
- *uncoverable*, if the assignments contained in the tuple tp clash with the constraints of the combinatorial problem.

The presented approach is implemented by Algorithm 4, and it is iteratively repeated for all the tuples and, at the end, a test is extracted from every test context (see Algorithm 5).

6.4.1 pMEDICI: exploiting MDDs for combinatorial test generation

In this section, I present the pMEDICI tool, which implements the structure shown in Fig. 37 by using Multi-valued Decision Diagrams (MDDs) as the underlying structure in each test context.

Background on MDDs

The tool presented in this section exploits *decision diagrams*, as defined in the following.

Definition 10 (Decision diagram)**.** A *decision diagram* is a graph that represents a function $f : D \rightarrow B$ where $D = D_1 \times \cdots \times \cdots D_n$ and B is the Boolean domain, i.e., $B = \{F, T\}$.

Typically, a decision diagram is used to evaluate the truth value of a function f when applied to variables $x_1, \cdots, x_n$. If all domains D_i are binary, then we use Binary Decision Diagrams (BDDs) to represent Boolean functions. Multivalued Decision Diagrams (MDDs), instead, extend BDDs by allowing every variable to have a different domain with a different number of elements. It is important for each MDD to respect the following properties:

- only two terminal nodes must be present, which are labeled as `F` and `T`;

Algorithm 4 Tuple consumption procedure.

```
Require: $TupBuffer$, the buffer containing the tuple already produced and ready to be consumed
Require: $TC$, the list of all the test contexts
Require: $M_C$, the CIT model

    ▷ Extract the tuple from the tuple buffer
 1: $t_j \leftarrow TupBuffer.extractFirst()$

    ▷ Try to find a test context that implies the tuple
 2: $tc \leftarrow findImplies(TC, t_j)$
 3: if $tc$ is not $NULL$ then
 4:     $t_j.setCovered()$
 5:     return
 6: end if

    ▷ Try to find a test context which is compatible with $t_j$
 7: $tc \leftarrow findCompatible(TC, t_j)$
 8: if $tc$ is not $NULL$ then
 9:     $tc.updateTC(t_j)$
10:     $t_j.setCovered()$
11:     return
12: end if

    ▷ Create a new empty test context
13: $tc \leftarrow createTestContext(M_C)$
14: $tc.addConstraints(M_C.getConstraintsList())$
15: if $tc.isCompatible(t_j)$ then
16:     $tc.updateTC(t_j)$
17:     $t_j.setCovered()$
18: else
19:     $t_j.setUncovered()$
20: end if
21: return
```

- every non-terminal node must be labeled by an input variable x_i and must have $|D_i|$ outgoing labeled edges, i.e., one per each possible value of the domain;
- every variable must appear only once in the MDD, in any path from the root to a terminal node;

Given these properties, an MDD can select which values of the domain D are accepted by the function f (i.e., which values lead to the terminal node `T`). In fact, if the values $x_1, \ldots, x_n$ for the variables in D are selected by f, then $f(x_1, \ldots, x_n) = T$, otherwise $f(x_1, \ldots, x_n) = F$.

Typically, among MDDs, unary operations can be performed (*complement*, or a computation of *cardinality* that represents the number of all possible paths leading to the terminal node `T`). The cardinality value of an MDD can be used to check the consistency between boolean functions. In fact,

Algorithm 5 Tests collection.

```
Require: $TC$, the list of all the test contexts
Require: $Threads$, the list of all the test builder threads
Require: $TupBuilder$, the thread building the tuples
Ensure: $TS$, the vector containing the test cases

   ▷ Join all the threads
1: for each $thread \in Threads$ do
2:     $thread.join()$
3: end for
4: $TupBuilder.join()$

   ▷ Gather all the tests generated by the test contexts
5: for each $tc \in TC$ do
6:     $TS.add(tc.getTest())$
7: end for
```

if $f_1(x)$ and $f_2(x)$ are inconsistent, the intersection between the MDDs representing the two functions is empty, for all the values of x. This is one of the properties exploited for generating combinatorial test suites with MDDs. Furthermore, with MDDs, the most classical binary operations such as *union*, *intersection*, and *difference* can be performed.

In particular, since MDDs can represent logic functions, operations among MDDs are equivalent to logical operations:

- given an MDD M representing the function f, the complement $\neg M$ represents the function $\neg f$;
- the union between two MDDs $M_1 \cup M_2$ represents the function $f_1 \vee f_2$;
- the intersection between two MDDs $M_1 \cap M_2$ represents the function $f_1 \wedge f_2$.

How to deal with combinatorial models with MDDs

As previously introduced, MDDs allow for the expression of Boolean functions. Hence, an MDD can be used to handle the boolean function computing the validity of the assignments to each parameter in a combinatorial model. In fact, if ignoring the constraints, a combinatorial model with n parameters, each one with cardinality p_i, can be easily represented using an MDD M_{TS} with n non-terminal nodes (each one associated with the name of the corresponding parameter) and with p_i outgoing labeled edges for all the parameters except the last one, which has only one edge connected to the terminal node T - since each assignment is valid if constraints are ignored. Fig. 38 represents an example of MDD M_{TS} modeling a simple combinatorial model with only three parameters, each with two possible values and without constraints. Every path from the root to the terminal `T` is a syntactically correct

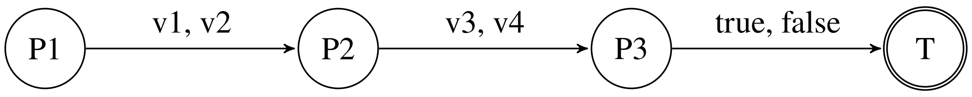


Figure 38: Example of MDD for a simple combinatorial model.

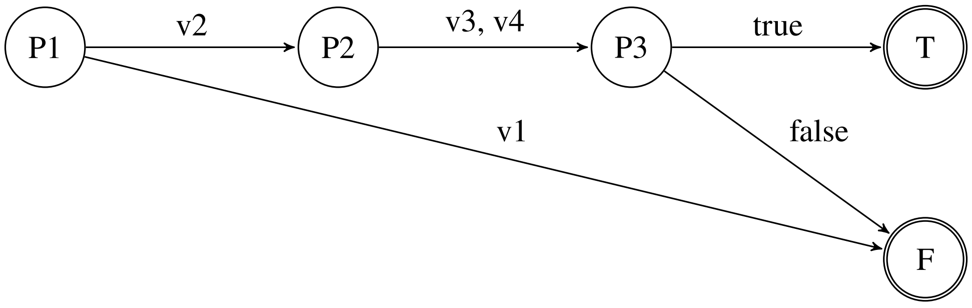


Figure 39: MDD structure when a constraint is included.

assignment of values to the parameters. Thus, from the MDD M_{TS} all tests that can be derived, i.e., all possible paths from the start to the terminal node, are valid tests. On the contrary, if also the constraints are considered, some path will lead to the terminal note F (i.e. those that will violate the constraints).

Every constraint can be represented as a Boolean formula containing the operators $\neg, \vee, \wedge$ and the equality between the parameters and their values. Therefore, each constraint can be represented by an MDD that models its truth function. In particular, the intersection between the MDD that models the parameters M_{TS} and the MDDs derived from each constraint is a new MDD that accepts only valid tests that comply with all constraints. For example, Fig. 39 shows how the MDD in Fig. 38 evolves when the constraint (P3 = true AND P1 = v2) is included.

Note that when a tuple needs to be added to an MDD, it can be considered as a "new constraint", so the operation previously described (intersection between the MDD representing the tuple and the one representing the system) can be performed. As I will discuss later in detail, pMEDICI builds the tests incrementally, by collecting suitable tuples in order to obtain valid tests: the intersection computation is iteratively repeated, in all the test contexts, for all the tuples.

The pMEDICI tool

pMEDICI[3] takes advantage of the concepts previously explained to generate combinatorial test suites. It integrates mddlib[4], by the Consortium for Logical Models and Tools, which is the only open-source Java library that natively supports MDDs and allows the computation of logical operations between them.

[3]pMEDICI is available online at `https://github.com/fmselab/ct-tools/tree/main/pMEDICI`

[4]`https://github.com/colomoto/mddlib`

The structure of the tool is shown in Fig. 37, where each X_i is, in this case, the MDD internal to each test context. Considering the MDD, and the process shown in Algorithm 4, given a tuple t_j to be handled by a test builder thread in pMEDICI:

- if the thread finds a test context tc_i in which the tuple t_j is already *implied*, t_j is consumed and marked as covered;
- if a thread finds a test context tc_i in which the tuple t_j is *compatible*, t_j is handled by tc_i, which updates its MDD structure by computing the intersection between the current MDD and the one corresponding to the tuple t_j. Then, t_j is consumed, and marked as covered;
- if a thread can not find a test context tc_i in which the tuple t_j is compatible or implied, a new test context is created together with its MDD structure that initially contains only the constraints of the combinatorial model. If t_j is compatible with the newly created test context, the tuple is consumed and marked as covered; otherwise, it means that the tuple is not compatible with the constraints, so it is marked as *uncoverable* and skipped.

During the test building procedure, pMEDICI adopts several optimizations:

- **Test context selection**: the threads building the tests can select the test context according to some policies (for instance, by giving precedence to those having some relations with the tuple to be added). In particular, in pMEDICI, test contexts are ordered in such a way that the first to be considered is the one that has the highest cardinality of the MDD (number of possible paths from the first parameter to the `true` leaf) after the addition of the considered tuple. The experiments carried out confirmed that this technique allows the tool to create fewer tests with higher variability. However, ordering the test contexts has proved not to be as time-effective as expected when working on models with a lot of parameters or constraints, since the time required for the ordering process overpass that of the test generation algorithm. For this reason, the optimization can be disabled by the user.
- **Test context management**: in multi-thread applications, the synchronization of threads while using shared objects is paramountly important. In the case of pMEDICI, test contexts are shared between test builder threads, so they need to be handled in mutual exclusion. However, only write instructions (insertion of a new tuple and intersection computation) are critical, so pMEDICI can be configured to lock the test contexts only when writing. Experiments carried out on this optimization have highlighted that if this optimization is active, the size of the test suites is consistently reduced. This optimization can be disabled by the user.

- **Management of the constraints**: test contexts could optimize the storage of constraints, or in case there are none, simplify the process. In pMEDICI, MDDs are not actually used by test contexts if no constraint is present in the combinatorial model. In that case, only the list of assignments A is updated. In this way, the tool avoids the computation of the intersection among MDDs (which is the most expensive operation in terms of time), and this allows saving time.

Limitations

The pMEDICI tool has some limitations that are directly related to the use of MDDs. Limits are mainly due to constraint management. In fact, pMEDICI is not able to deal with models containing constraints with:

- Arithmetical operators, such as $+$, $-$, $\cdot$ and $/$;
- Comparisons between parameters, such as $p_1 = p_2$ or $p_1 \neq p_2$;
- Relational expression between parameters, or between a parameter and a value, such as $p_1 > 2$.

Although in principle MDDs could also deal with arithmetic and variable comparison by converting the constraints to pure Boolean expressions, in practice it is not easy to be done and risks generating constraints with exponential length. In addition, using MDDs requires each constraint in the combinatorial model to be converted in RPN (Reverse Polish Notation), which may decrease the readability of the models.

Results and comparison with ACTS

In this section, I present the results obtained when generating test suites for the MVM case study (whose combinatorial model is shown in Listing 6.1) using pMEDICI and compare them with those obtained with ACTS. Note that it is a rather simple model and only represents limited features of the system. For example, no numerical configuration parameters and their bounds are modeled since, as presented before, pMEDICI cannot deal with constraints containing relational expressions.
Tab. 25 shows the results obtained when comparing pMEDICI and ACTS in the MVM case study. They have been obtained on a PC with Windows 11, Intel i7-3770 with 3.4 GHz, 8 threads, and 32 GB RAM. Note that the results reported in Tab. 25 are the average of 10 executions.
For the analyzed case study, the test suites obtained always have 8 tests in all configurations and for all tools. However, for other combinatorial models, pMEDICI has generally demostrated to produce larger test suites than ACTS. This is the price of having multiple threads working on test generation, since more test contexts (and test cases) may be created. More details about these experiments are available in [33].

		Optimizations			
	N Threads	**TC Selection**	**TC Management**	**Size**	**Time [ms]**
pMEDICI	1	NO	NO	8	178
	1	NO	YES	8	171
	1	YES	NO	8	180
	1	YES	YES	8	175
	2	NO	NO	8	175
	2	NO	YES	8	168
	2	YES	NO	8	173
	2	YES	YES	8	174
	4	NO	NO	8	90
	4	NO	YES	8	84
	4	YES	NO	8	95
	4	YES	YES	8	87
	8	NO	NO	8	78
	8	NO	YES	8	72
	8	YES	NO	8	80
	8	YES	YES	8	76
ACTS	1	-	-	8	2939

Table 25: Comparison between pMEDICI and ACTS in the MVM case study.

On the other hand, pMEDICI always produces test suites in a significantly shorter time than ACTS. The time reduction, when pMEDICI is used instead of ACTS, is in most cases greater than 97%. It can be concluded that the use of pMEDICI for complex systems, such as PEMS, allows for faster test generation and a consistent reduction in the cost of testing. In this way, more time resources may be available for the testing process and better-quality devices may be obtained.

In terms of the number of threads, the results obtained show that increasing the number of threads (up to the limit of the threads supported by the CPU architecture) allows for decreasing the test suite generation time. Note that other experiments have shown that, on models with fewer parameters and constraints than the MVM, it is not always a good choice to increase the number of threads, as having more threads means adding coordination effort (which may be useless for simple models). More details about these experiments are available in [33].

Finally, for the MVM case study, the most effective optimization has proven to be related to text context management. On the other hand, using the test context ordering does not allow for reaching the same results. In fact, the analyzed model has too many parameters and constraints, and the effort required to order the test context list overpasses that of the test generation itself.

6.4.2 KALI: exploiting SMT solvers for combinatorial test generation

In Sect. 6.4.1, I have presented a tool that implements the structure described in Fig. 37 and manages the constraints and tests using MDDs. However, despite the overall optimal performance shown in the experiments performed in the MVM case study, pMEDICI still has many limitations due to MDDs. For this reason, in this section, I present KALI, which substitutes the MDD with an SMT solver and allows for managing all types of constraints and parameters.

Background on SMT solvers

Satisfiability modulo theories (SMT) are commonly used for generalizing, by using first-order formulas, Boolean satisfiability (SAT) and several additional features, such as equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories. These theories are handled by an SMT solver, which is a tool for deciding the satisfiability (or validity) of formulas in one or more theories. In research, SMT solvers are used for several tasks, e.g., the extended static checking, predicate abstraction, test case generation, theorem proving, and bounded model checking over infinite domains, to mention a few.
Many solvers are available in the literature, such as Yices [73], Z3 [69], SMTInterpol [60] and MathSAT5 [62]. All these solvers are capable not only of checking if a formula is satisfied or not but also of generating a model that satisfies the SMT formula represented in context. For this reason, SMT solvers are commonly used, as is done in KALI, for test generation [144]: after having represented all the parameters and constraint in the SMT context, the model generated by the SMT solver is the valid test we are looking for.

How to represent CIT models, constraints, tuples, and tests with SMT

In the following, the encoding of parameters, constraints, and tuples in the SMT solver is described.

- **Parameters encoding**: When an SMT logical context is defined, the parameters of a combinatorial problem can be represented as variables in that context. However, since not all SMT solvers support the same type of variables, the translation from combinatorial parameters to SMT variables has to be done accordingly to the type:
  - *Boolean parameters* are represented as simple SMT Boolean variables;
  - *Integer ranges* are represented as SMT integer variables. Nevertheless, in combinatorial problems the concept of “integer” is not defined as in SMT: only ranges are allowed.

For this reason, when converting a range to an SMT variable, an additional constraint specifying the lower and upper bounds of the range must be added. For example, if a range is defined in the combinatorial model as `P1 : [0 .. 3]`, in addition to the `P1` integer variable, the following constraint is added: $P1 \geq 0$ `AND` $P1 \leq 3$;

– *Enumerative parameters* are the most critical ones since not all the SMT solvers support them. In order to propose a solution that can be adapted to all SMT solvers, KALI converts each enumeration into a set of SMT Boolean variables and adds a set of constraints assuring that only one of the possible Boolean variables must be true. This process is generally defined as *flattening* [97]. For example, if an enumeration is defined in the combinatorial model as `P2 : {v1 v2 v3}`, it is translated into three different SMT Boolean variables `P2v1`, `P2v2`, and `P2v3`. In addition, the following constraint are added to the SMT context: `P2v1 <=> (not P2v2 and not P2v3)`, `P2v2 <=> (not P2v1 and not P2v3)`, and `P2v3 <=> (not P2v1 and not P2v2)`.

- **Model constraints**: The constraints of a combinatorial problem can easily be mapped to the SMT constraints, exploiting the previously defined variables. In particular, a combinatorial model may contain different types of constraints, such as relational, mathematical, or comparison operators (between parameters or values) in general propositional formulas. All these types of constraints can be easily represented with operations between variables and values defined in an SMT context.

- **Tuples**: Given a tuple tp, it can be represented in the logical context of SMT by simply adding a new constraint that limits the values of the parameters included in the tuple to the same values specified. For example, a tuple $tp = \langle P1, v1\rangle\langle P2, v2\rangle$ is translated in the following SMT constraint $P1 = v1$ `AND` $P2 = v2$.

Note that the mapping proposed between combinatorial models and SMT solvers resolves all the limitations of the approach proposed by pMEDICI in Sect. 6.4.1:

- Arithmetical operators are supported as part of formulas by all SMT solvers;
- The comparison between parameters is translated into a comparison between SMT variables;
- Relational expressions are supported by all SMT solvers when defining formulas.

For this reason, KALI (like all other tools based on SMT solvers) can deal with all types of combinatorial model, parameters, and constraints that can be present in a combinatorial problem.

The KALI tool

The KALI tool takes advantage of the concepts previously explained to generate combinatorial test suites. It integrates the java-smt library[5] and is available at `https://github.com/fmselab/ct-tools/tree/main/KALI/code`.
The structure of the tool is shown in Fig. 37, where each X_i is, in this case, the logical SMT context assigned to each test context. Considering the process implemented by KALI and shown in Algorithm 4, given a tuple t_j to be handled by the test builder threads:

- the function `findImplies` at line 2 extracts from the list of all test contexts TC the first test context tc_j that already implies the considered tuple tp_i, if it is present. Then, if tc_j is found, tp_i is consumed and marked as covered;
- the function `findCompatible` at line 7 extracts from the list of all test contexts TC the first test context tc_j which is compatible with the tuple tp_i (i.e., the tuple does not clash with the assignments already committed to the test context and with the constraints of the combinatorial model), if it is present. Then, if tc_j is found, tp_i is handled by tc_j, which updates its internal SMT logical context, consumes tp_i, and marks the tuple as covered;
- if no test context tc_j in which the tuple tp_i is compatible or implied is found, a new test context is created. It is initialized by the function `createTestContext` (line 13) which builds a new logical context for SMT and creates all the variables and constraints of the combinatorial problem in tc_j. After creating the new test context, if tp_i is compatible with it, the tuple is consumed, added to the context, and marked as covered; otherwise, it means that the tuple is not compatible with the constraints, it is marked as *uncoverable* and skipped, and the empty test context is discarded.

Since KALI shares the same structure as pMEDICI and only substitutes the MDDs with SMT solvers, during test generation the same optimizations are available. Additionally, KALI implements a fourth optimization which allows the user to select the order in which the tuple generator thread generates the tuples. The available ordering options are:

- `IN_ORDER_SIZE_DESC` (OD), which first generates the tuples starting from the parameters assuming the highest number of values;
- `IN_ORDER_SIZE_ASC` (OA), which first generates the tuples starting from the parameters assuming the lowest number of values;

[5]`https://github.com/sosy-lab/java-smt`

	N Threads	Size	Time [ms]
KALI	8	8	2082
pMEDICI	8	8	72
ACTS	1	8	2939

Table 26: Comparison between KALI, pMEDICI, and ACTS on the MVM case study.

- `RANDOM` (RD), which shuffles the list of parameters before starting the tuple generation process;
- `AS_DECLARED` (AD), which considers the parameters in the order in which they are declared within the combinatorial model. This is the same approach used by pMEDICI.

Results and comparison with pMEDICI and ACTS

After having described KALI, its implementation, and the optimizations that are embedded in the tool, this section presents the results obtained when applying KALI to the MVM case study reported in Listing 6.1.

Tab. 26 reports these results and the comparison with those obtained with the best configuration of pMEDICI and ACTS in the MVM case study. They have been obtained on a PC with Windows 11, Intel i7-3770 with 3.4 GHz, 8 threads, and 32 GB RAM. Note that, like the ones presented for pMEDICI, the results reported in Tab. 26 are the average of 10 executions. Moreover, for KALI, here, I report only the results obtained with the best configuration possible (the same number of threads as the one supported by the CPU, the SMTInterpol solver, the ordering optimization for test contexts enabled, the OD parameter ordering, and locking the test contexts only when needed for writing). Experiments that have been carried out to identify this configuration are reported in [35].

The results obtained confirm that using multi-thread for the MVM case study allows for reducing the test suite generation time (KALI is faster than ACTS), and it does not influence the test suite size. However, in other experiments, KALI has generally shown to produce larger test suites than ACTS and pMEDICI (see experiments in [35]), and not to be the best performing even in terms of test suite generation time. Indeed, also in the MVM case study, using pMEDICI is always the best choice.

The performance of KALI is deeply influenced by the powerful but heavy structure of the SMT solvers that underlie the tool. In fact, the experiments clearly indicate that using the approach implemented by KALI comes with a price. However, we can identify two main advantages w.r.t. other tools and algorithms:

- pMEDICI outperforms KALI, but it cannot handle complex constraints that cannot be represented in MDDs, as previously reported. In fact, the MVM model reported in Listing 6.1 is a simplified version of the mechanical ventilator. In practice, other aspects that parameterize the behavior of the mechanical ventilator should be included in the combinatorial model. For example, a more realistic model should include, among others, the following parameters:

```
InspiratoryPressure: [0 .. 100]
MaxInspPressure : [0 .. 100]
```

These two parameters represent the inspiratory pressure set by the doctor and the maximum inspiratory pressure that cannot be exceeded without triggering an alarm. For this reason, an additional constraint should be added:

```
# InspiratoryPressure <= MaxInspPressure #
```

In this case, pMEDICI could not have been used, since it is not able to deal with the added constraint, while ACTS or KALI can deal with it. Given its multi-thread nature, KALI performs better than ACTS on this model.

- ACTS supports a large set of constraints, and it is very efficient. However, ACTS and similar approaches build the whole test suite by adding a parameter one by one, and this means that no test is complete until the generation is finished. KALI, instead, uses a one-test-at-a-time strategy, and after a short period of time, a test case is already available. This makes approaches like KALI more suitable for online testing [171], where test execution begins during test generation. In this case, having tests immediately available can reduce the time required to discover faults and can drive test generation to improve the fault detection capability.

6.5 Conclusion

In this chapter, I have presented the concepts of Combinatorial Interaction Testing (CIT) and how this test generation strategy can be applied to medical systems. As reported in the literature, and proved also in the MVM case study in this chapter, the use of combinatorial testing can aid testers in decreasing the number of test cases needed to identify the same number of faults.
CIT is usually used to test systems by varying the input values. However, in this chapter, I have presented an approach, based on automata, that exploits combinatorial testing in order to test event-based systems and builds test sequences composed of events that are sequentially executed. Typical examples of event-based systems in the medical domain are all protocols that are used for communication among medical devices. Sect. 6.2 reports the application of the combinatorial automata-based approach for sequence generation to the PHD protocol case study and demonstrates how it allows for reaching a

higher coverage w.r.t. the one obtained with methods classically adopted for sequence testing based on SCAs.
Furthermore, the research of a method for benchmarking combinatorial test generators (now embedded into the CTWedge environment) has permitted identifying a function defining the cost of adopting a specific test generator instead of another. By considering this function, in order to reduce the cost of test generation, multi-threaded techniques have been presented in Sect. 6.4 and implemented in the pMEDICI and KALI tools. The former uses MDDs for managing constraints and combinatorial models, while the latter substitutes MDDs with SMT solvers, in order to overcome the limitations of MDDs while handling particular types of constraints. The results obtained, presented in this chapter, have shown how the generation time of the test suite can be significantly reduced when using KALI or pMEDICI instead of cutting-edge tools such as ACTS on rather complex medical systems such as the MVM. For this reason, since a consistent saving in time is obtained, the additional time may be used to further investigate and improve the quality of the software products.

Part III

Robustness for AI-based medical software

Chapter 7. Neural network robustness

In the previous chapters, good software engineering principles have been outlined, and model-based methods have been analyzed to investigate their applicability to PEMS. However, recent medical software and devices, such as PEMS, are even more dependent on AI components, which allow the estimation of medical quantities or the diagnosis of pathologies. Thus, assessing the safety and reliability of these components, such as neural networks (NNs), used for medical purposes is of paramount importance, as well as trying to increase their dependability.

Normally, software testing and good software engineering practices (such as those described in the previous chapters) can be used for regular software if the safety of the device embedding the software has to be ensured. However, even when considering the guidelines presented in Chapter 3, having an NN-based system may change the way in which the activities have to be performed. For example, considering a medical device embedding a NN, it could be a good choice to isolate it into a separate component, which does not depend on others, and to test it independently. Anyway, classical testing approaches used for standard software are not suitable when NNs are present in a system, because the behavior of a NN can not always be properly forecasted and, moreover, it may be difficult even to define test goals which may include uncertainty [77]. Nevertheless, the basic idea behind the testing remains the same: the software should be robust enough when dealing with not-correct inputs. Thus, NN-based medical software should be validated as well as regular software, and, in this chapter, I will present the approaches I devised for this purpose.

This chapter is based on the work published in [3–6] and is structured as follows. Sect. 7.1 presents the general state of the art on neural network robustness. In Sect. 7.2, I present the background concepts on which the proposed definition of robustness, introduced in Sect. 7.3, is based. In Sect. 7.4, a tool for automatically computing the robustness of a neural network and a method to minimize the time required for its computation are presented. Sect. 7.5 presents methods suitable for increasing the robustness of a neural network. Finally, Sect. 7.6 presents how robustness relates to medical devices and their certification, while Sect. 7.7 concludes the chapter.

7.1 State of the art on neural network robustness

Testing machine learning models has recently become a hot topic that has been defined over several properties, e.g., correctness, robustness, and fairness [183]. However, as previously introduced, it is difficult to test machine learning applications using software testing techniques originally designed for code [152]. For this reason, most of the research work focuses on analyzing behavioral properties (such as *robustness*) for neural networks instead of trying to "test" them as done for the code.

Most of the papers regarding the robustness of neural networks focus on the *adversarial examples*, i.e., inputs specifically created with the aim of fooling a neural network, resulting in the misclassification or misinterpretation of a given input. Normally, these inputs are indistinguishable from the human eye, but can cause the network to fail to correctly interpret them. A well-known example is the one in Fig. 40: if a small noise, not visible to the human eye, is added to an image of a panda, the neural network used for classifying the pictures changes its classification into a gibbon. Adversarial examples can be created using different types of attack, called *adversarial attacks*, e.g. the fast gradient sign method attack, which is a white-box attack whose goal is to ensure misclassification. A common approach in assessing the robustness of a NN w.r.t. adversarial examples is the one presented in [175] where a theoretical analysis of this type of robustness is given. The authors propose a method to find the key reasons why an adversarial example can fool a classifier and consider these oracles during the network training phase to make it immune to that specific kind of adversarial example. Another typical approach for evaluating adversarial robustness is the one proposed by [78], where adversarial examples are simulated by using semi-random noise, which has been shown to generalize adversarial examples in a simple way.

Note that in some application scenarios, such as in the medical domain, adversarial examples are unlikely to occur [123] and so different types of robustness should be used. Moreover, Carlini and Wagner [56] demonstrate that some of the most recent techniques used to increase the robustness w.r.t. adversarial examples can be easily fooled with other adversarial generation techniques. Thus, when developing an NN, especially for safety-critical domains, it is of paramount importance to evaluate and increase the robustness for plausible alterations as well, depending on the application domain, as proposed in this chapter.

In the literature, other papers investigating the robustness of neural networks without considering adversarial examples are available, but none of them proposes a formal definition of *robustness* as, instead, is presented in this chapter. For example, a study of CNN robustness to appearance variability in biomedical images is presented [166]. The authors introduce a new type of layer, called *neighborhood similarity layer* (NSL), to improve the robustness w.r.t. changes in the appearance of objects that are not well represented by the training data. In [41], CNN-Cert, a general framework capable of certifying the robustness of general convolutional neural networks, is presented. It is able to deal with CNNs composed of convolutional layers, max-pooling layers, batch normalization layers, residual blocks, as well as general activation functions. A methodology similar to that used in this

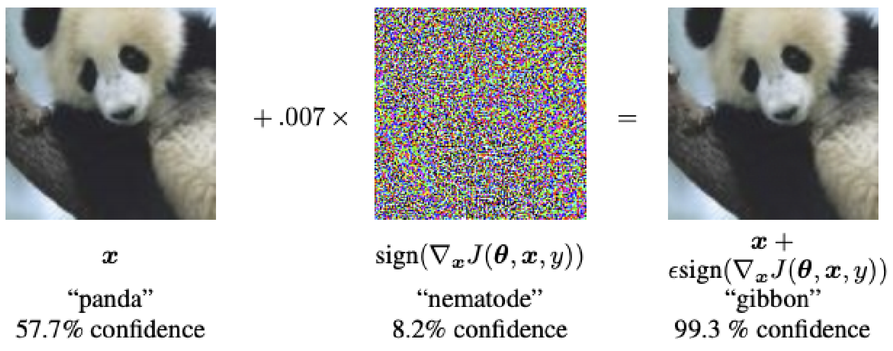

+ .007 ×
=
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 40: Adversarial examples.

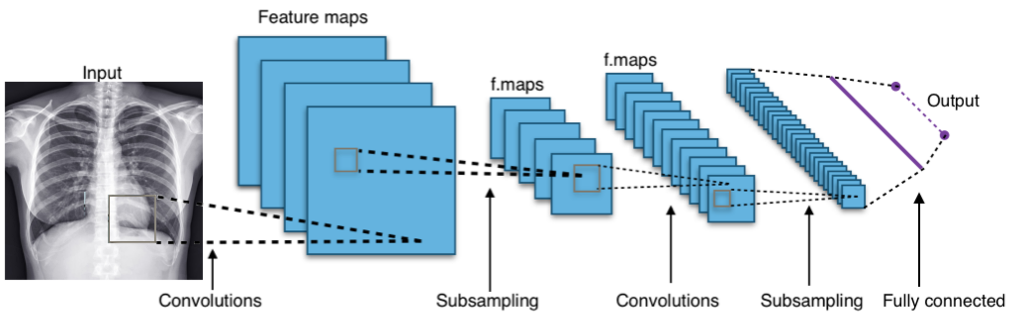


Figure 41: Typical structure of a Convolutional Neural Network.

chapter is used in image manipulation detection [111], but, also in this case, a formal definition of robustness has not been introduced.

7.2 Background concepts

In this section, basic concepts useful for understanding the idea of neural network robustness are presented.

7.2.1 Types of neural networks typically used in PEMS

Artificial Neural Networks (ANNs) are used in many critical tasks in the medical domain, among which classification or estimation. For example, Convolutional Neural Networks (CNNs) can be used to analyze medical images, e.g., coming from medical exams, with the aim of diagnosing possible diseases. An example of a CNN is shown in Fig. 41: through convolutional and subsampling layers, parts of an input are analyzed to identify relevant features. On the other hand, Multilayer Perceptrons (MLPs) can be used when a medical quantity (such as blood oxygen pressure or patterns in the respiration waveforms) needs to be estimated or predicted. This can happen when no sensor measuring the quantity exists or when the quantity needs to be derived from other data. Fig.42 shows an example of the structure of an MLP.

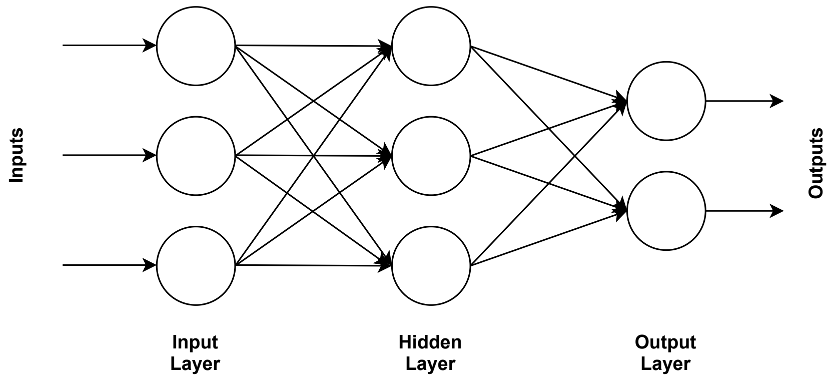


Figure 42: Typical structure of a Multilayer Perceptron.

In this chapter, ANNs trained to be used as classifiers, i.e., to assign a label (taken from a set of possible categories) to an input (e.g., an image, a sound, a text, etc.) or as estimators are analyzed. If we consider a generic ANN that receives an input $t \in P$, where p is the *input space*, the two types of neural network can be defined as follows:

Definition 11 (Classifier)**.** A *classifier* C for inputs t can be seen as a function that assigns a label l to an input $t \in P$, i.e., $C(t) = l$.

Definition 12 (Estimator)**.** An *estimator* E for inputs $\bar{t}$ can be seen as a function that computes the output o for each $\bar{t} \in P$, i.e., $E(\bar{t}) = o$.

7.2.2 Accuracy and errors

As previously introduced, in the medical domain, an NN can be used as an estimator or a classifier. In both cases, the networks' results are subject to errors or misclassifications. In general, the lower the error or misclassification, the higher the quality of the NN.
In particular, if the NN is used as a classifier, its quality can be measured in terms of *accuracy*:

Definition 13 (Accuracy)**.** The *accuracy* of a classifier C w.r.t. a set of inputs P is defined as the ratio of correctly evaluated inputs in P, i.e.,

$$acc(C, P) = \frac{|\{t \in P \mid C(p) = label(p)\}|}{|P|}$$

where *label* gives the correct evaluation of an input t.

On the other hand, if the NN is used as an estimator/predictor, the main approaches to evaluate its performance are *Mean Squared Error* (MSE) and *Mean Absolute Error* (MAE). The former represents the average of the squared difference between the target value and the value estimated by the model; since it squares the residuals, it penalizes even small errors, leading to an overestimation of how bad the model is. The latter is the absolute difference between the target value and the value estimated by the model. Given this definition, MAE is more robust to outliers and does not penalize errors as much as MSE [177]. However, since the MAE scale is the same as the data being measured, its value is absolute, and it is difficult to easily understand the relative error. For this reason, when analyzing the performance of an estimator, it is better to use *Mean Absolute Percentage Error* (MAPE), i.e., the percentage equivalent of MAE.

Definition 14 (*MAPE*)**.** Considering an input set of size n, where y_i is the real value of the input t and $\hat{y}_i$ is the estimated value of the same input, MAPE is defined as follows:

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{\|y_i - \hat{y}_i\|}{y_i}$$

NN estimators can be compared using *MAPE* under nominal conditions ($MAPE_0$), which is calculated on the test set. Note that the MAPE metric, as warned in [98], is not applicable in problems where the real value of y_i is close to or equal to zero, because it results in very large numbers.

7.2.3 Alterations

When ANNs are used in practice, normally, they work in an environment that may be subject to unexpected external factors. For example, if CNNs are used for classifying medical images, it may happen that the inputs they are given are slightly different from those we may obtain under nominal conditions. In fact, the camera used for their acquisition might produce out-of-focus images or with a higher/lower brightness. The same may happen if an NN is used for estimation: the input signals may be disturbed by unforeseen electromagnetic interferences.
In particular, it is plausible that given a NN (either used as an estimator/predictor or as a classifier), by altering an input t, the trustworthiness of the response of the network will change and it will likely decrease when the alteration level increases. Therefore, the accuracy of the classifier or *MAPE* of the estimator also depend on the "quality" of the inputs used during their training and testing. For example, Fig. 43 shows how the accuracy of a NN classifier diminishes when changing the brightness of the images given as inputs. How can we define and measure the robustness of a NN used in the medical field when an alteration occurs, considering that, as presented in Sect. 7.1, the one w.r.t. adversarial examples is not meaningful while using NNs in healthcare processes? First, here, I define what can be considered an alteration.

Definition 15 (Alteration)**.** An *alteration* of type A of an input t is a transformation of t that mimics the possible effect on t when a problem occurs in reality during its acquisition or in its elaboration. In

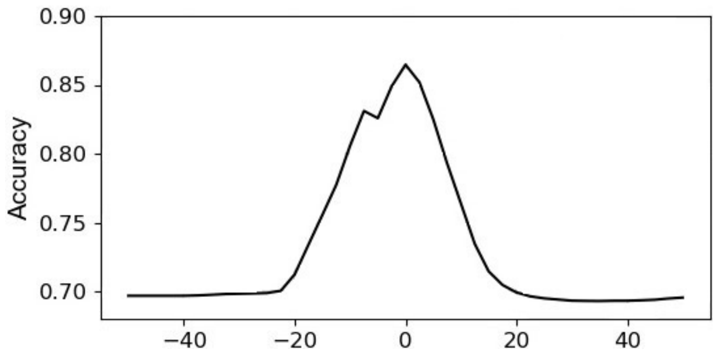


Figure 43: Accuracy change when brightness is altered in the inputs given to a CNN for medical image classification.

the following, I identify with P^{A_l} the set of data obtained by altering all the input data in P with an alteration of type A of *level* $l \in [L_A, U_A]$, where $[L_A, U_A]$ is the range of plausible alteration levels of type A.

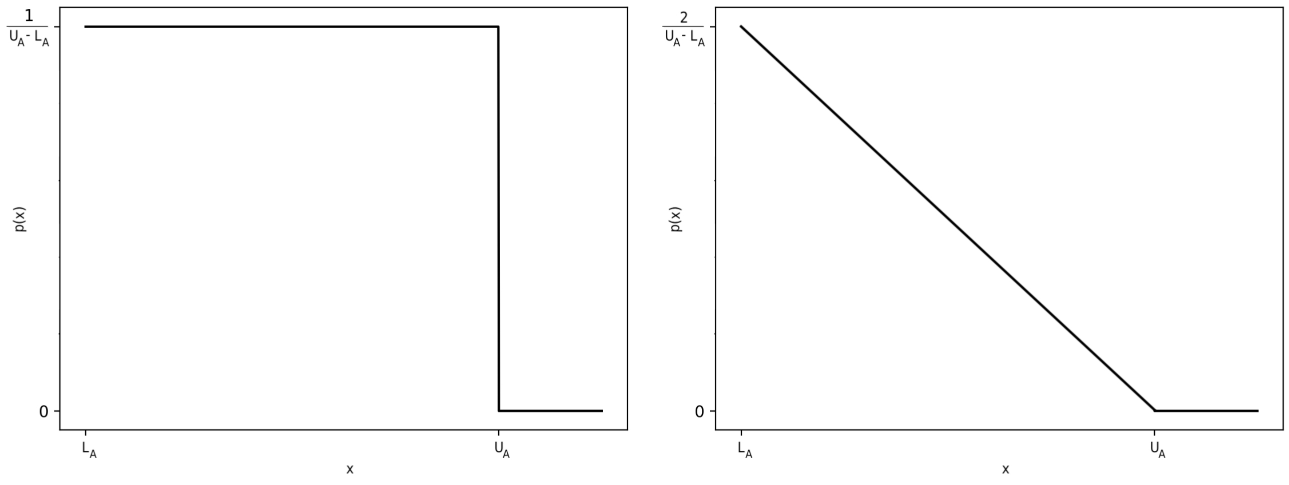
In some application domains, alterations can occur randomly at any level $l \in [L_A, U_A]$, while, for other types of applications, some alteration levels may be more likely to occur than others. To reflect this characteristic of alterations, it is possible to define the *alteration probability* as follows.

Definition 16 (Alteration Probability)**.** Given an alteration of type A, the *probability* of the alteration A, identified as p_A, is the probability distribution of the alteration levels having as support the interval $[L_A, U_A]$.

The most common examples of probability distributions for alterations are as follows:

- *Uniform probability*: all the alteration levels are equally probable, as shown in Fig. 44a and formally defined as:
$$p_A(x) = \begin{cases} \dfrac{1}{U_A - L_A} & L_A \leq x \leq U_A \\ 0 & \text{otherwise} \end{cases}$$
- *Linear probability*: lower alteration levels are more probable than higher levels, as shown in Fig. 44b. It is formally defined as:
$$p_A(x) = \begin{cases} \dfrac{2}{(U_A - L_A)^2} \cdot (U_A - x) & L_A \leq x \leq U_A \\ 0 & \text{otherwise} \end{cases}$$

However, other types of probability function can also be used, such as truncated normal or half-normal distributions, depending on the application domain.



(a) Uniform probability. **(b) Linear probability.**

Figure 44: Examples of functions describing the probability p_A of an alteration level A.

Note that the alteration probability can also be used to specify the "importance" the user wants to give to a level of alteration. For instance, a uniform probability is more likely to be used for systems that should be equally resilient to all levels of an alteration within a given interval. On the other hand, a linear probability can be preferred when the system is not critical, and it is more important to perform better on lower alterations than on the higher ones.

7.3 The general concept of NN robustness

Having presented in Def. 15 what here is considered an alteration, in the following I introduce the concept of robustness for NNs, both for estimators and classifiers. In both cases, the idea is that the quality measure (accuracy for classifiers and *MAPE* for estimators) should stay above a defined threshold for the highest number of alteration intensities possible.

7.3.1 Robustness for classifiers

When it comes to classifiers, as explained above, it is possible to measure their quality using their accuracy. In particular, given an alteration aA, the robustness of a NN classifier can be defined as follows.

Definition 17 (Robustness for NN classifiers)**.** Let Θ be a threshold representing the minimum accepted accuracy. The *robustness* of a classifier C w.r.t. alteration of type A in the range $[L_A, U_A]$ (using a set of inputs P) is defined as the percentage of the alteration values for which the accuracy is greater than

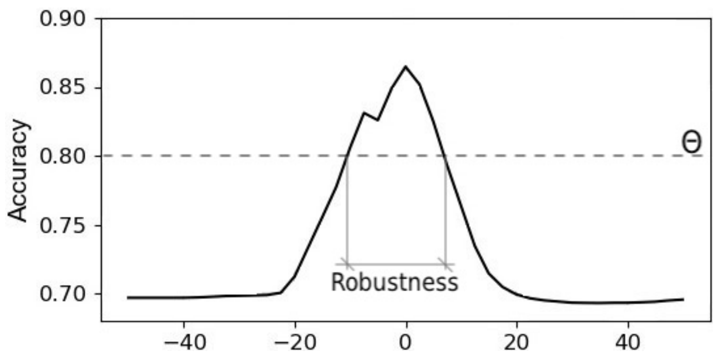


Figure 45: **Robustness computation for a classifier where a brightness alteration is applied as per Fig. 43.**

Θ. Formally:

$$rob_A(C,P) = \frac{\int_{L_A}^{U_A} H(acc(C,P^{A_i})-\Theta)di}{U_A-L_A} \cdot 100\% \quad \text{where} \quad H(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Fig. 45 shows the robustness of a classifier w.r.t. brightness alteration with $\Theta = 80\%$.
In most cases, accurately computing the robustness using this formula (where H is also known as the Heaviside function) is very difficult. In fact, the accuracy function is not known a priori and, therefore, should be computed for all alteration levels l in $[L_A, U_A]$, which could be many, if not infinite. Moreover, computing the accuracy of the network C when a single alteration level A_l is applied to the set P, requires considerable effort, so a method is necessary to select suitable alterations to reduce the computation time.
A naive solution is to uniformly sample in $[L_A, U_A]$ and compute the accuracy only for the points sampled. In this case, it is possible to count how many points (n_{acc}) the accuracy is acceptable, relatively to the total number of points sampled n. Formally:

Definition 18 (Uniform robustness)**.** Given n equi-distributed points $SP = \{l_1, \ldots, l_n\}$ sampled in the interval $[L_A, U_A]$, the *uniform robustness* is defined as:

$$rob_A(C,P) = \frac{n_{acc}}{n} \cdot 100\% = \frac{|\{l \in SP \mid acc(C, P^{A_l}) \geq \Theta\}|}{|SP|} \cdot 100\% \qquad (7.1)$$

Note that the previous definitions use accuracy to compute the robustness of a classifier. However, the definitions could be adapted to use recall, precision, or F1-score, depending on the context.

The limits of the uniform robustness definition

Robustness, as defined in Def. 18, may still require a lot of computational power and time to be evaluated, especially when the levels to be applied for each alteration are many. In fact, be n the number of alteration levels to be applied for an alteration A, k the number of inputs, and t_A the time

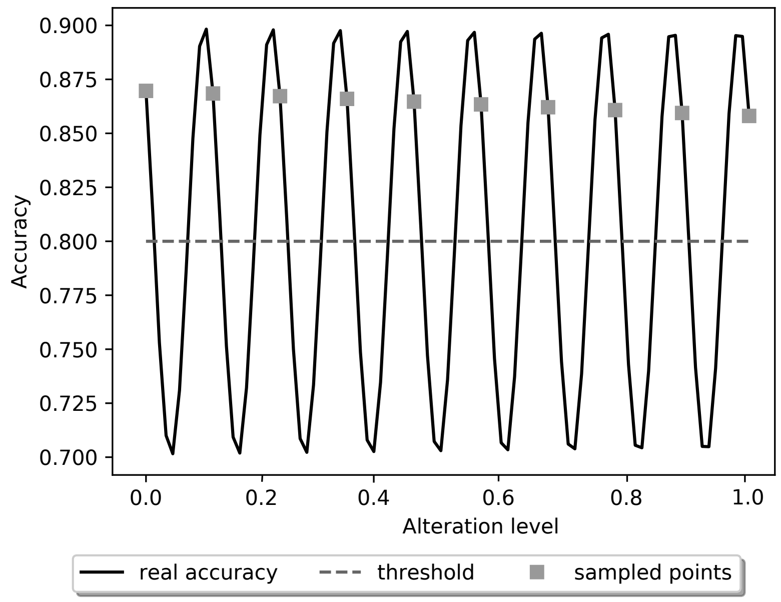


Figure 46: Error in robustness computation using a low number of uniformly distributed sampled points.

required to apply the alteration A to a single input, the total time required to perform robustness analysis is:

$$t_{tot_A} = n \times k \times t_A \tag{7.2}$$

For example, if we consider $n = 1000$, $k = 1000$, and $t_A = 0.1 sec$, the total time required to compute the robustness will be approximately 28 hours.

Several approaches can be used to decrease the time required for robustness analysis. Among them, the easiest and most effective ones are the followings:

- Reduction of the value n, i.e., the number of levels sampled for the alteration A. This is a viable solution but has some drawbacks, especially when analyzing networks whose accuracy varies a lot. For example, Fig. 46 shows the points evaluated when uniform sampling of an accuracy function is performed with $n = 10$. Using Def. 18, a robustness of $rob_A(C, P) = 100\%$ would be calculated. Nonetheless, the real value of the robustness corresponding to the analyzed accuracy function is significantly lower ($\simeq 50\%$).

- Adaptive selection of the points to be sampled (possibly not uniformly), as usually done for values in software testing. In fact, while testing a regular program, choosing the correct input parameters and the correct values is challenging because different inputs or values may lead to different bug discoveries. However, as in software testing, sampling some inputs is required, since exhaustive testing cannot be performed. This approach will be presented and discussed in Sect. 7.4.2.

Robustness and adversariability

As presented in Sect. 7.1, current research papers on neural network robustness mainly focus on adversarial robustness. However, in this chapter, only plausible alterations are considered for computing the robustness of NN classifiers. How does the robustness proposed in this chapter relate to the one w.r.t. adversarial robustness? In order to investigate this relation, here I discuss the notion of *adversariability*, in the case of classifiers for medical images. Nevertheless, similar considerations can be extended to other types of neural networks and inputs as well.
Unlike alterations (as defined in Def. 15), the generation of adversarial examples does not directly provide a measure of the difference between the starting input and the modified one. Therefore, the definition of adversariability is based on the classical definition of structural similarity index taken from [52] and defined as follows.

Definition 19 (Structural similarity index)**.** The *structural similarity index* between the two images p and q is defined as:

$$S(p,q) = \frac{(2\mu_p\mu_q + c_1)(2\sigma_{pq} + c_2)}{(\mu_p^2 + \mu_q^2 + c_1)(\sigma_p^2 + \sigma_q^2 + c_2)} \in [0,1]$$

where μ_p and μ_q are the averages of the pixel values in p and q, σ_p^2 and σ_q^2 are the variances of p and q, σ_{pq} is the covariance of p and q, c_1 and c_2 are two constants, and $S(p,q) = 1$ when p and q are identical.

Let $ADVEX(C,p)$ be the set of all adversarial examples generated by a given technique for a classifier C and be p an input image. $ADVEX(C,p)$ is empty only if p cannot be modified in a way to mislead C or the generation technique of *ADVEX* is not powerful enough. Every technique can generate many adversarial examples. For example, if an image is manipulated for obtaining an adversarial example, higher levels of manipulation will likely lead to other adversarial examples as well. Among all adversarial examples, the concept of *adversariability* is defined over the *most adversarial* one:

Definition 20 (Most adversarial example)**.** Let C be a binary classifier, and p be an correctly classified image, that is, $C(p) = label(p)$. The *most adversarial example* is defined as the most similar image to p that is misclassified (if it exists), formally:

$$p^{ae} = \underset{p' \in ADVEX(C,p)}{\arg\max}\; S(p,p')$$

Note that $ADVEX(C,p)$ may be empty. In this case, we say that p^{ae} does not exist.

Using Def. 19 and 20, it is possible to define the *adversariability*, i.e., the vulnerability w.r.t. adversarial examples, as follows.

Definition 21 (Adversariability)**.** Let C be a binary classifier and P be a set of inputs. The *adversariability* of C is defined as the percentage of input $p \in P$ correctly evaluated for which there exists an adversarial example p^{ae}, *weighted* by the similarity index between p and p^{ae}. Formally:

$$adv(C, P) = \frac{\sum_{p \in CE} \hat{S}(p, p^{ae})}{|CE|} \in [0, 1]$$

where $\hat{S}$ is equal to the similarity index S if the adversarial example p^{ae} exists, 0 otherwise; and $CE = \{p \in P \mid C(p) = label(p)\}$ is the subset of P of correctly classified inputs.

Higher adversariability values mean that the classifier C is more vulnerable to adversarial examples. Note that adversarial examples p^{ae} that are more similar to the original image p (i.e., those having higher similarity index $\hat{S}(p, p^{ae})$) are those that contribute the most to the adversariability: indeed, they represent the most insidious cases in which an imperceptible modification misleads the classification.

7.3.2 Robustness for estimators

When dealing with estimators, their quality is usually measured using *MAPE*. In fact, it can be logical to consider that, when an alteration is applied, the *MAPE* of the estimator will change and, in particular, it may increase when the level of the alteration applied increases. However, different values of *MAPE* can be more or less acceptable, depending on the criticality of the task performed by the NN and the precision required by the requirements of the system. Therefore, the *tolerance* to the NN error can be defined as follows:

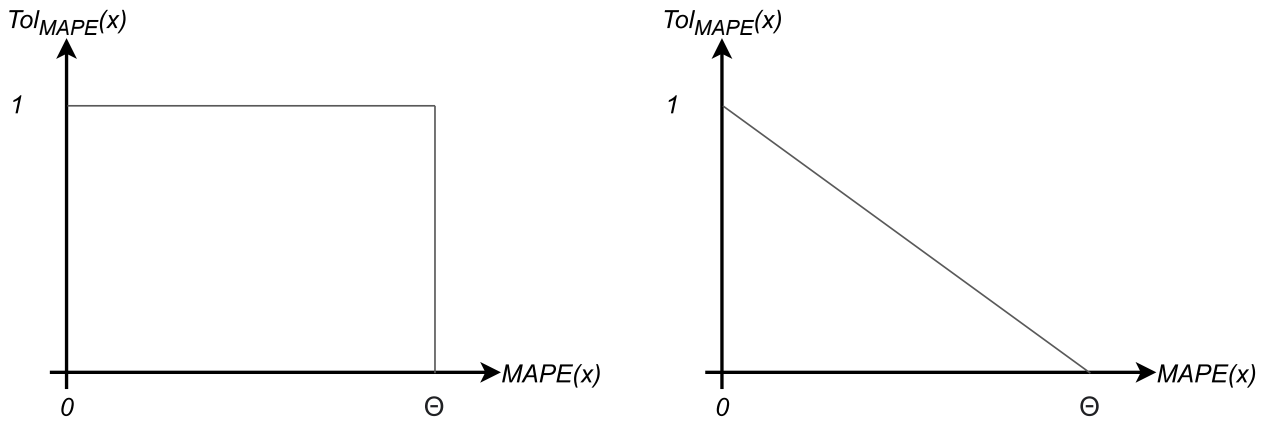
Definition 22 (Tolerance)**.** Let Θ be a threshold representing the maximum *MAPE* value accepted by the system requirements and $MAPE_A(x)$ the error value when an alteration A of level x is applied to the input data. The desired *tolerance* for the error $MAPE_A(x)$ is a function $Tol_{MAPE_A}(x)$ such that:

$$\begin{aligned}
&Tol_{MAPE_A}(x) = 1 && \text{for } MAPE_A(x) = 0\\
&0 \leq Tol_{MAPE_A}(x) \leq 1 && \text{for } 0 < MAPE_A(x) \leq \Theta\\
&Tol_{MAPE_A}(x) = 0 && \text{for } MAPE_A(x) > \Theta
\end{aligned}$$

The type of tolerance function can be chosen by users depending on the application domain. In the following, two examples of tolerance functions are described:

- *Uniform tolerance*: all the different values of $MAPE_A(x)$ are tolerated in an equal way, as shown in Fig. 47a. It is formally defined as:

$$Tol_{MAPE_A}(x) = H(\Theta - MAPE_A(x)) \text{where } H(k) = \begin{cases} 1, & k \geq 0 \\ 0, & k < 0 \end{cases}$$



(a) Uniform tolerance. **(b) Linear tolerance.**

Figure 47: Examples of functions describing the tolerance.

The intuition is that, as long as *MAPE* is below or equal to the threshold Θ, the tolerance is maximum, otherwise it is 0.

- *Linear tolerance*: lower values of $MAPE_A(x)$ are more tolerated than higher values, as shown in Fig. 47b. It is formally defined as:

$$Tol_{MAPE_A}(x) = \frac{\max\left(\Theta - MAPE_A(x), 0\right)}{\Theta}$$

Given the tolerance function *Tol*, the *alterations A* and their *probability p_A*, the *robustness* for NN estimators is defined as follows.

Definition 23 (Robustness of NN estimators)**.** Let E be an NN estimator under evaluation, $MAPE_A(x)$ be the value of the error done by E when an alteration A of level x is applied to the input data, $p_A(x)$ the probability of the alteration, and $Tol_{MAPE_A}(x)$ the tolerance for *MAPE* values of the selected network. The *robustness $rob_A(E) \in [0, 1]$* of E w.r.t. alterations of type A in the range $[L_A, U_A]$ is formally defined as:

$$rob_A(E) = \int_{L_A}^{U_A} Tol_{MAPE_A}(x) \cdot p_A(x)\ dx \cdot 100\% \tag{7.3}$$

Intuitively, the robustness can be seen as the sum (integral) of all the errors the network commits when all possible alterations are applied. Alterations are weighted by their probability and errors by the specified tolerance.
Note that the robustness depends on the type of tolerance and probability chosen. For example, it is possible to define the following sub-types of robustness:

- **UL robustness**: It is obtained when uniform probability (any alteration level is equally likely) and linear tolerance (lower errors are preferable) are chosen. In particular, the following formula computes the robustness:

$$rob_A^{UL}(E) = \frac{\int_{L_A}^{U_A} \max\left(\Theta - MAPE_A(x), 0\right)\, dx}{\Theta \cdot (U_A - L_A)} \cdot 100\%$$

This definition of robustness evaluates the ratio between the striped red area and the gray one in Fig. 48a. It can be used for systems where, for higher alteration values, large *MAPE* values are acceptable, while, for lower alteration values, the smaller the *MAPE*, the better.

- **LU robustness**: It is obtained when linear probability and uniform tolerance are chosen. In particular, the following formula computes the robustness:

$$rob_A^{LU}(E) = \frac{\int_{L_A}^{U_A} H(\Theta - MAPE_A(x)) \cdot (U_A - x)\, dx}{\frac{1}{2} \cdot (U_A - L_A)^2} \cdot 100\% = \frac{\int_{x \in [L_A, U_A] | MAPE(x) < \Theta} \left(\Theta \cdot \frac{U_A - x}{U_A - L_A}\right) dx}{\frac{1}{2} \cdot \Theta \cdot (U_A - L_A)} \cdot 100\%$$

  This definition of robustness evaluates the ratio between the area of the striped red region and the area of the gray triangle in Fig. 48b. The definition is suitable for systems where it is crucial to respect the threshold Θ along the entire alteration interval $[L_A, U_A]$, in particular for low (and more likely) levels of alteration.

- **UU robustness**: It is obtained when uniform probability and uniform tolerance are used:

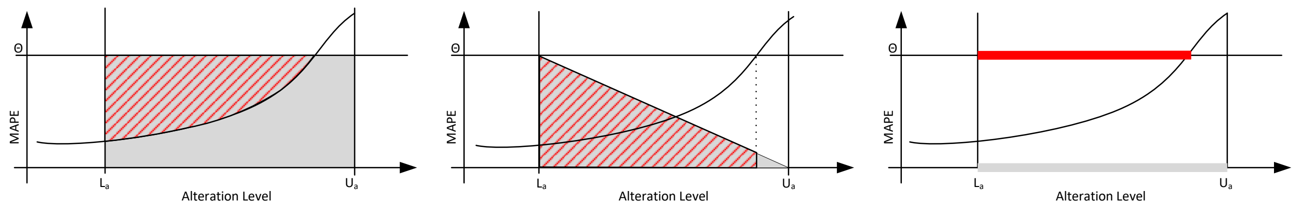
$$rob_A^{UU}(E) = \frac{\int_{L_A}^{U_A} H(\Theta - MAPE_A(x))\, dx}{\Theta \cdot (U_A - L_A)} \cdot 100\%$$

  This definition computes the ratio between the lengths of the red and gray lines in Fig. 48c. UU robustness is suitable for systems where it is crucial to respect the threshold Θ throughout the alteration interval $[L_A, U_A]$, regardless of the probability of alteration, such as in the case of medical devices.

- **LL robustness**: It is obtained when linear probability and linear tolerance are used:

$$rob_A^{LL}(E) = \frac{\int_{L_A}^{U_A} \max(\Theta - MAPE_A(x), 0) \cdot (U_A - x)\, dx}{\frac{1}{2} \cdot \Theta \cdot (U_A - L_A)^2} \cdot 100\%$$

  Higher levels of alteration, which have a greater impact on the input data, may lead to higher values of *MAPE*. However, in some systems, these alteration levels may be less probable than the lower ones, and users may be less worried about some high error in very rare cases. Thus,



(a) UL robustness. **(b) LU robustness.** **(c) UU robustness.**

Figure 48: Graphical representation of different types of robustness.

LL robustness is suitable for these kinds of system, when the user does not want to penalize too much high error values for the highest alteration levels. Note that it is difficult to provide a graphical interpretation of LL robustness as done for the other types of robustness.

7.3.3 Properties of the robustness measure

The robustness definition, either for classifiers or estimators, guarantees the following properties:

1. the robustness of a model M is always between 0 and 1, i.e., $0 \leq rob_A(M) \leq 1$;
2. if a network has always zero error, its robustness is 1, i.e., $rob_A(M) = 1$ if $\forall x \in [L_A, U_A]$, $MAPE_A(x) = 0$ in the case of estimators, or $\forall x \in [L_A, U_A]$, $acc(C, P^{A_i}) = 100\%$ in the case of classifiers. Note that the condition is sufficient but not necessary, i.e., the robustness can also be 1 for systems in which the error is greater than 0 for some alteration level (e.g., in estimators when uniform tolerance is used and the error is never greater than Θ);
3. if a network has an error always greater than the specified threshold Θ, its robustness is 0: $rob_A(M) = 0$ if $\forall x \in [L_A, U_A]$, $MAPE_A(x) > \Theta$ in the case of estimators, or if $\forall x \in [L_A, U_A]$, $acc(C, P^{A_i}) < \Theta$ in the case of classifiers.

7.4 Tools and algorithms for robustness estimation of NN classifiers

As explained in Sect. 7.3, computing the robustness of a NN is a viable technique to analyze its quality. However, computing this measure may take a long time, especially in the case of classifiers, since the classification process is normally more expensive than the estimation one (for example, it may need to deal with pictures instead of numbers). Moreover, configuring an environment allowing the robustness analysis can be a complex activity since it requires the definition of alterations, the computation of the robustness formula, etc. Thus, in the following, I introduce ROBY, a software allowing the automatic computation of robustness for NN classifiers, and ASAP, a technique aiming at reducing the time required for robustness estimation, without losing estimation precision.

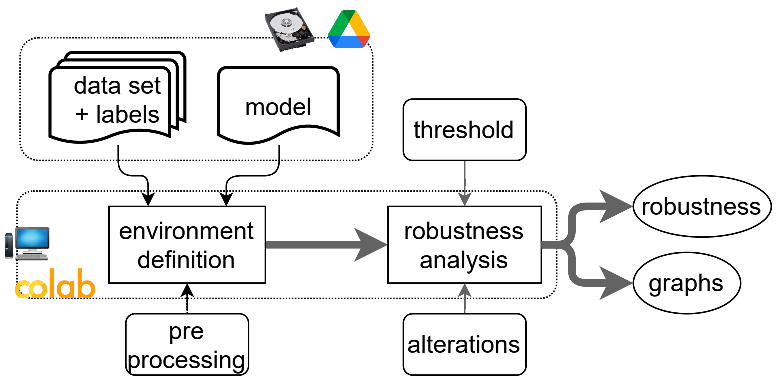


Figure 49: ROBY workflow.

7.4.1 ROBY

In this section, I present ROBY, a Python tool for ROBustness analYsis. The tool has been engineered so that it can be used, with minimal effort, by different users in different domains and for different types of data when dealing with classifiers. A user must only specify:

- the location of the test data set;
- a labeling function or a label list, allowing the tool to retrieve the correct classification of the test input data;
- which alterations have to be applied to input data (either the standard ones provided by the tool, or custom-made);
- where to run the robustness computation (either locally or on Google Colab).

As a result, ROBY computes the robustness measure for the different alterations and produces plots that visualize how the accuracy changes when the alterations are applied. In this way, users may observe which are the most critical alteration levels and intervene directly on them. ROBY evaluates the robustness as reported in Def. 18. The tool is available at `https://github.com/fmselab/roby` and can also be installed using the pip package manager. Before using ROBY, users must make sure that their system satisfies some requirement:

- The model must represent a classifier and must be written in a format supported by Keras, i.e., `HDF5` or `SavedModel`;
- All the inputs in the test data set must be previously labeled and must be expressible in the `np.ndarray` format, a rather general format in which images, audio, text, and video can be represented;
- Each alteration must be expressible as an input modification between a minimum and maximum threshold.

Fig. 49 shows the workflow to follow when using ROBY to analyze the robustness of an NN classifier. In particular, the following activities need to be performed by the user:

1. Supply a data set, together with the labels (possibly given as a *function*, `labeler`, that is applicable to each input data and gives as output the correct label) and a *model*. Data sets and models can be stored either on the local hard drive (if the robustness computation is performed locally) or on Google Drive (if the computation is performed on Google Colab);
2. Define the *environment* that may optionally include, besides the input data together with their labels and the model, also the *pre-processing* function;
3. Select suitable *alterations* among those provided by ROBY or define new ones in accordance with the domain;
4. Specify the desired *threshold* Θ to be used for computing the robustness;
5. Run ROBY which computes the robustness measure and produces plots showing how the accuracy of the model changes w.r.t. different levels of alterations.

How to use ROBY in custom domains

Each domain may have different thresholds, alterations, and data-type to be considered. For this reason, ROBY allows users to define a method to load the data set, a method to adapt to different data formats, and another method to assign labels to input data. Moreover, users can define custom alterations and use them to evaluate the robustness of the NN under analysis with ROBY. These *extension points* are better detailed below.

- **Data loading**: ROBY works for ANNs used as classifiers receiving `np.ndarray` input data. Users can create a custom testing environment `EnvironmentRTest` by giving either the paths for all input data or a list of data already in array format. In the former case (when paths are given), the user must specify the way to be used to convert the file data into the `np.ndarray` format by declaring a `reader(file_name)` function.

- **Data labeling**: correct labels for input data can be given with a list of all labels or by using a `labeler` function. In the former case, the list must be of the same size as the data set, while in the latter case, the user has to define a function that receives input data and returns a string representing the real label for the selected input.

- **Custom alterations**: ROBY has an embedded abstract class `Alteration` that can be easily extended to create custom alterations. When extending the abstract class, the user is requested to implement the functions `name()`, to return the name of the alteration, and `apply_alteration(data, alteration_level)`. Given the input data in `np.ndarray`, the function `apply_alteration` returns the data (still in the same format) with the applied alteration of the desired level. Moreover, ROBY supports the definition of `AlterationSequence`, which can be used to represent an alteration caused by the composition of multiple alterations.

- **Pre-processing**: In some domains, ANNs could have been trained with data of shapes, sizes, or formats different from those used for testing. For these reasons, during the declaration of the testing environment `EnvironmentRTest`, users can specify an additional pre-processing function. It is applied to each input data, after an alteration is applied, before its recognition by the ANN. Typical pre-processing functions are used when there is the need to remove, e.g., white borders from images, resize or extract a relevant part from the input, or clip the volume of an audio track.

7.4.2 The ASAP algorithm

In order to tackle the limitations of the uniform sampling approach (previously explained in Sect. 7.3.1), here I present the *ASAP* (Adaptive SAmpling by Parabolic estimation) algorithm to automatically select the points where to evaluate the accuracy. It is based on the assumption that the best points to select would be those in which the accuracy curve intersects the threshold Θ. However, since the analytical form of the accuracy function is not known a priori and it is not possible to compute these intersections, users should try to select points as close as possible to Θ. ASAP is based on a parabolic approximation of the accuracy curve: once the user has computed the accuracy for two alteration levels A and B, the real accuracy curve between A and B will be included in the area between two parabolas passing through the points A and B, and having concavity depth respectively $+\hat{a}$ and $-\hat{a}$ (see Fig. 50), i.e., with equation $y = ax^2 + bx + c$ with $a = \pm\hat{a}$. If there is an intersection between the area marked in Fig. 50 and the threshold Θ, and the distance between A and B is sufficiently large (I will discuss later what can be considered "sufficiently large"), then the accuracy of the middle point M between A and B has to be computed. After that, the point M is added to the sample set and the procedure is applied recursively to the two intervals $[A, M]$ and $[M, B]$. In this way, the number of points evaluated is adaptively determined and depends both on the value of the parameter $\hat{a}$ and on the behavior of the accuracy function. Intuitively, the higher the value of $\hat{a}$, the higher the number of alteration levels evaluated by the algorithm. For this reason, users should choose the $\hat{a}$ value based on the precision required and on the time available for robustness analysis.

Algorithm 6 describes how the approximation method works. It recursively considers two alteration

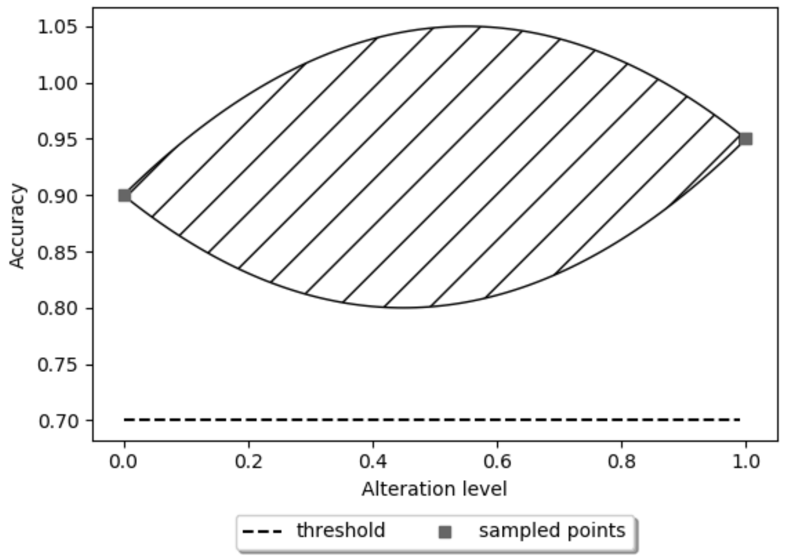
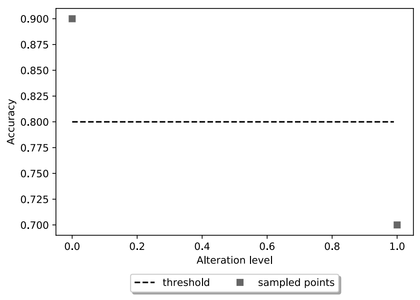
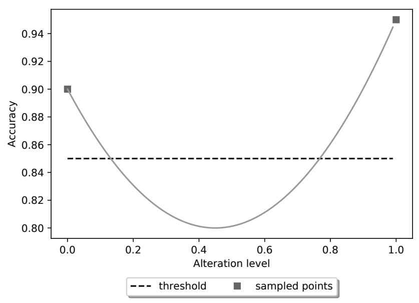


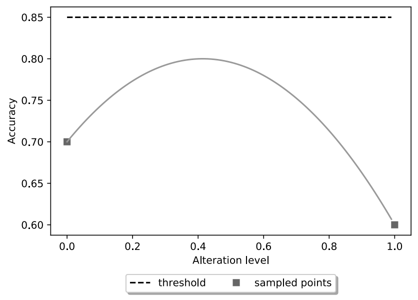
Figure 50: Area in which the real accuracy curve between *A* and *B* will be likely included, identified by two parabolas with concavity depth $\pm\hat{a}$.



(a) *A* is above $y = \Theta$ and *B* under $y = \Theta$.



(b) The parabola with concavity depth $\hat{a}$ intersects with y=Θ.



(c) The parabola with concavity depth -$\hat{a}$ does not intersect with y=Θ.

Figure 51: Different positions of points during recursive accuracy evaluation.

levels x_A and x_B in $[L_A, U_A]$, and evaluates the accuracy of the model in them, using the function *getAccuracy* that applies the selected level of alteration (lines 2-3) to the test set *TS*.

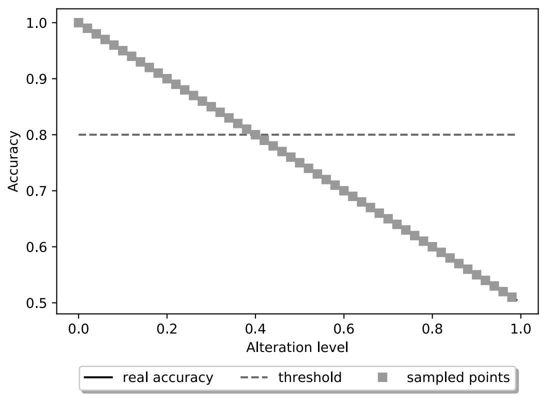
Then, ASAP checks, by using the function *parabIntsct* (line 4), whether at least one of the two parabolas passing for the two points x_A and x_B intersects the threshold. A sufficient condition is that the two accuracy values are opposite w.r.t. the threshold Θ (see an example in Fig. 51a): this is checked at line 11. If this is not the case, i.e., both accuracy values are above or below the threshold Θ (see examples in Figs. 51b-51c), the algorithm computes the parabolas passing for *A* and *B* and having concavity depth $\pm\hat{a}$, using the *parabola* function (line 14). Note that given the concavity depth $\hat{a}$, the coefficients *b* and *c* must also be calculated. They are obtained by solving the following system of equations (where $a = \pm\hat{a}$ is fixed by the user):

$$\begin{cases} a \cdot {x_A}^2 + b \cdot x_A + c = acc_A \\ a \cdot {x_B}^2 + b \cdot x_B + c = acc_B \end{cases}$$

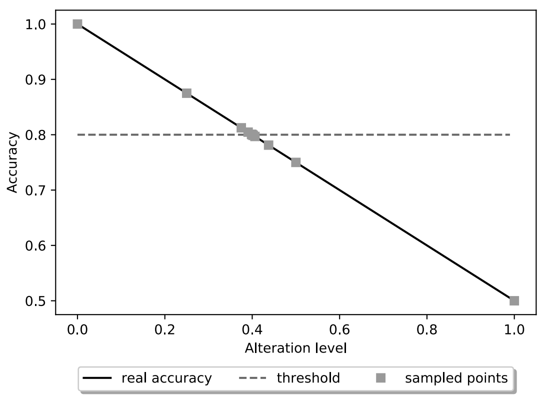
Algorithm 6 ASAP: Adaptive SAmpling by Parabolic estimation.

```
Require: $x_A$ the first alteration level
Require: $x_B$ the second alteration level
Require: $TS$ the test set including all the input data (e.g. images)
Require: $\Theta$ the threshold to be used for robustness analysis
Require: $\hat{a}$ the concavity depth parameter to be used by ASAP
Require: $minStep$ the minimum step between two alteration levels
Require: $C$ the CNN to be analyzed
Ensure: $RES$ the list of sampled points with their accuracy values
1: procedure EVAL($x_A$, $x_B$, $TS$, $\Theta$, $\hat{a}$, $minStep$, $RES$, $C$)
2:     $acc_A \leftarrow getAccuracy(TS, x_A, C, RES)$        ▷ Get the accuracy in $A$
3:     $acc_B \leftarrow getAccuracy(TS, x_B, C, RES)$        ▷ Get the accuracy in $B$
       ▷ Check whether the parabolas with concavity depth $\pm\hat{a}$ intersect $\Theta$
4:     $intersected \leftarrow parabIntsct(x_A, acc_A, x_B, acc_B, \hat{a}, \Theta) \lor parabIntsct(x_A, acc_A, x_B, acc_B, -\hat{a}, \Theta)$
       ▷ Estimate accuracy in the two sub-intervals if they are not too close
5:     if $intersected \land x_B - x_A \geq minStep$ then
6:         EVAL($x_A$, $\frac{x_A+x_B}{2}$, $TS$, $\Theta$, $\hat{a}$, $minStep$, $RES$, $C$)
7:         EVAL($\frac{x_A+x_B}{2}$, $x_B$, $TS$, $\Theta$, $\hat{a}$, $minStep$, $RES$, $C$)
8:     end if
9: end procedure

10: function $parabIntsct$($x_A$, $acc_A$, $x_B$, $acc_B$, a, $\Theta$)
    ▷ Check whether $A$ and $B$ are opposite w.r.t. $\Theta$
11:     if $(acc_A - \Theta) \cdot (acc_B - \Theta) < 0$ then
12:         return $true$
13:     end if
        ▷ Compute the parabola and its vertex
14:     $b, c \leftarrow parabola(x_A, acc_A, x_B, acc_B, a)$
15:     $(x_v, y_v) = (-\frac{b}{2 \cdot a}, -\frac{b^2 - 4 \cdot a \cdot c}{4 \cdot a})$
16:     return $(x_A \leq x_v \leq x_B) \land ((acc_A - \Theta) \cdot (y_v - \Theta) < 0)$
17: end function

18: function $getAccuracy$($TS$, $x$, $C$, $RES$)
19:     if $\neg RES.contains(x)$ then
20:         $acc_x \leftarrow ComputeAccuracy(TS, x, C)$
21:         $RES.append(\langle x, acc_x \rangle)$        ▷ Save the obtained results
22:     else
23:         $acc_x \leftarrow RES.get(x)$
24:     end if
25:     return $acc_x$
26: end function
```

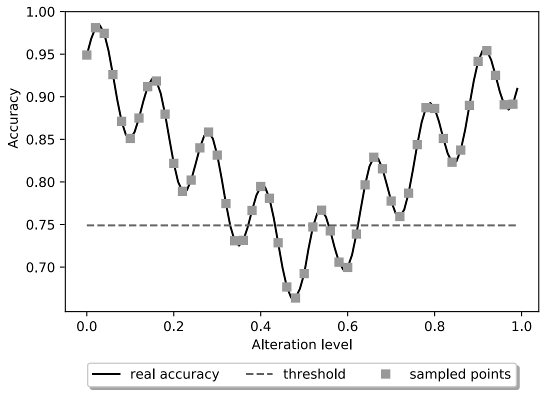
Then, the parabola vertex $V(x_v, y_v)$ is calculated (line 15). The method verifies that the parabola is in the area of interest (line 16), by checking that: (i) $x_v \in [x_A, x_B]$ (first operand of the conjunction), and (ii) the parabola intersects the threshold Θ, i.e., y_v is opposite to acc_A w.r.t. Θ (second operand). This process is repeated for both $a = \hat{a}$ and $a = -\hat{a}$.



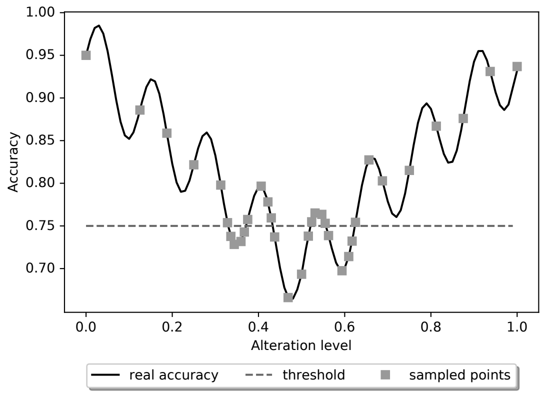
(a) Uniform sampling - Linear function.



(b) ASAP - Linear function.



(c) Uniform sampling - Irregular function.



(d) ASAP - Irregular function.

Figure 52: Examples of robustness computation using synthetic functions.

Finally, if one of the two parabolas intersects the threshold and the sampled points are not too close (line 5), the computation is recursively repeated in the intervals $[x_A, x_M]$ and $[x_M, x_B]$ (lines 6-7), where x_M is the average alteration level between x_A and x_B. Note that the user must choose the value $minStep$ coherently, taking into account possible time constraints, in order to obtain a correct robustness estimation.

In this way, a set of sampled points is obtained, each with the corresponding accuracy: $RES = \{\langle l_1, acc_1\rangle, \ldots, \langle l_n, acc_n\rangle\}$. Starting from RES, the robustness is computed by generalizing the formula in Def. 18 as follows:

$$rob_A(C,P) = \frac{\sum_{j=2}^{n} H(acc_j - \Theta) \cdot (l_j - l_{j-1})}{U_A - L_A} \cdot 100\%$$

Example of robustness computation using ASAP

Fig. 52 reports the effect of ASAP on two different synthetic functions. In particular, in Fig. 52a, the robustness calculation is performed by using uniform sampling with 50 equidistributed alteration levels, and a robustness of 38.8% is obtained, while in Fig. 52b only 15 levels are used by ASAP (with $\hat{a}$ equal 256) and a robustness of 39.06% is obtained. Since the curve is generated synthetically, the real robustness can be computed, and it turned out to be 40.00%. Note that the two results are very close to each other and also close to the real robustness. In this particular case, ASAP is able to perform even better than the normal approach with uniform sampling, despite fewer sampled points being used.

The same behavior can be observed by comparing Fig. 52c, where robustness 77.9% is obtained by uniform sampling with $n = 50$, with Fig. 52d, where ASAP uses only 34 alteration levels (focused in the area near the threshold value instead of uniformly distributed ones), obtaining a robustness of 81.2%. Note that even in this case the two results are close and that the real robustness associated with the accuracy plot shown in Fig. 52c and Fig. 52d is 80.4%. This shows that ASAP uses fewer alteration levels, so it saves time but still provides an accurate approximation of the robustness.

Maximum error estimation of the computed robustness

ASAP exploits a parabola-based approximation of the accuracy curve for a neural network, so the estimation provided may be subject to errors. However, it provides theoretical guarantees regarding the maximum error that it can make in computing the robustness. To define this, pairs of two consecutive points p_j and p_{j+1} must be selected from *RES*, so that the parabolas passing from them with concavity depth $\pm\hat{a}$ intersect the threshold Θ, i.e.,

$$IP = \left\{ (l_j, l_{j+1}) \;\middle|\; \begin{array}{l} \langle l_j, acc_j\rangle, \langle l_{j+1}, acc_{j+1}\rangle \in RES \wedge \\ \left(\begin{array}{l} parabIntsct(l_j, acc_j, l_{j+1}, acc_{j+1}, \hat{a}, \Theta) \vee \\ parabIntsct(l_j, acc_j, l_{j+1}, acc_{j+1}, -\hat{a}, \Theta) \end{array} \right) \end{array} \right\}$$

Intuitively, each pair of points p_j and p_{j+1} identifies the points between which at least one of the two parabolas with concavity depth $\pm\hat{a}$ intersects Θ, and, therefore, also the real accuracy curve may intersect, but ASAP has quit sampling because the two points have alteration levels sufficiently close w.r.t. to the chosen $minStep$.

Under the assumption that the user has chosen an appropriate value for $\hat{a}$, the error that ASAP can commit only comes from the intervals identified in *IP*. This intuition is formalized by the following theorem.

Theorem 1. *Let C be a CNN and A an alteration defined in the range $[L_A, U_A]$. Let rob_A be the robustness computed for C and A by ASAP using a given $\hat{a}$. Let rob_A^O be the* real *robustness value. Under the assumption that $\hat{a}$ is a suitable parameter, i.e., the real accuracy curve is included in the areas of two parabolas with concavity depth $\hat{a}$ (see Fig. 50), the maximum error of the computed* *robustness has a guaranteed upper bound defined as follows:*

$$|rob_A - rob_A^O| \leq \varepsilon_A \quad with\ \varepsilon_A = \frac{\sum_{(l_j, l_{j+1}) \in IP} (l_{j+1} - l_j)}{|U_A - L_A|} \tag{7.4}$$

Proof. The error in robustness computation is due to the cases in which the real curve crosses the threshold line but ASAP fails to find the exact intersection point. Let's consider where this can happen by considering all the sub-intervals $[l_j, l_{j+1}]$ of the points in *RES*:

- if $(l_j, l_{j+1}) \notin IP$, then the parabolas with concavity depth $\pm\hat{a}$ do not intersect the threshold. Since, by the assumption of the theorem, $\hat{a}$ is a suitable parameter, the real curve is included in the computed parabolas, and so it also does not intersect the threshold. So, no contribution of error in robustness computation comes from these points.
- if $(l_j, l_{j+1}) \in IP$, then we can distinguish two cases:
  - the two points are opposite w.r.t. the threshold line. So, the real curve intersects the threshold line, but in an unknown point that does not belong to *RES*.
  - the two points are both below or above the threshold: ASAP ignores the possible intersection of the real curve with the threshold line since, for ASAP, the sampled points are close enough.

  In both cases, the maximum absolute error is $l_{j+1} - l_j$.

Therefore, the total error in the robustness estimation is given by the sum of errors for all pairs of points in *IP*. Hence, the upper bound of the error is as defined in Eq. 7.4. □

7.5 How to improve the robustness of a NN

After having analyzed the robustness of a NN one may want to apply techniques allowing the improvement of the network in terms of its robustness w.r.t. plausible alterations, in particular in safety-critical systems (as in the PEMS domain) which should be as robust as possible.

Obviously, the first applicable solution is to create a more complex CNN, which is able to guarantee higher robustness; however, this solution could be too costly and, moreover, the designer does not have any hint on how to modify the network in order to increase its robustness. Therefore, in the following, I consider additions to the training data or automatic extensions of the network that do not require the intervention of the designer since they do not require the development of a new NN.

Note that a good technique should not only improve robustness, but also not degrade the classification of the unaltered input; therefore, while trying to enhance robustness, one should also check the accuracy of the retrained network.

7.5.1 Data augmentation

Data augmentation is a wide and well-known subject [169], including a suite of techniques that increase the size and quality of the training data set. Using data augmentation, the NN has to be retrained with a new training set composed of the original training set and additional data that can be created starting from the original ones, so that the network performance, obtained after the retraining process, is enhanced.

Since data augmentation is proposed here as a method for increasing the robustness of a NN, the new training set may be created in two different ways, depending on the network type and on the application domain:

- With *recombined data*: the idea of this approach is to create new input data (virtual) by recombining existing ones (real) [71].
- With *altered data*: the idea behind this approach is that the robustness of a NN w.r.t. plausible alterations (as per Def. 15) may be increased by adding in the training set a certain number of inputs that have been altered with the same alterations under test.

Data augmentation has been shown to be very effective in the literature. However, it requires complete retraining of the whole network, which has been performed using a large number of input data. Thus, a lot of time should be spent in order to apply this technique.

7.5.2 Incremental learning

Data augmentation contributes to enhancing the robustness performance of a network, but requires the retraining of the original network with a larger set of inputs, including the altered ones. Thus, to reduce the training effort and increase the generalization of the network at the same time, other techniques may be used. In particular, a different approach, known as *incremental learning* is suitable in most cases, as it allows one to improve the performance of the model without retraining the entire network. In the literature, it is performed whenever new samples are available, by adjusting what has been learned according to them. This method has been designed to work as an online technique, but in the analysis presented in this book, it has been adapted (as previously suggested by [161]) to be used as an offline approach.

This approach adds knowledge to an existing NN without modifying it in order to improve its performance. For this reason, a new NN composed of two sub-networks (as shown in Fig. 53) is created: one sub-NN represents the original network, and the other one the new and non-previously trained

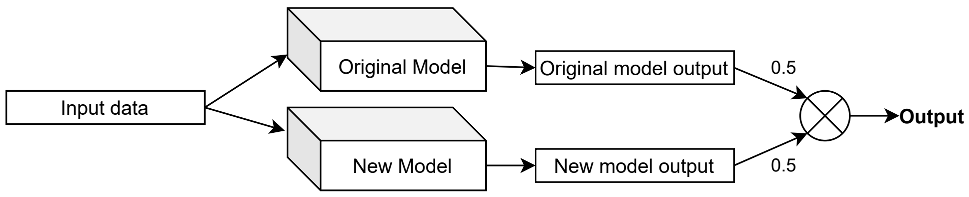


Figure 53: Incremental learning technique.

network (but still with the same network structure as the original one). During the retraining phase, only the new sub-network is trained using the new altered data set; the outputs of the two networks are then used in the loss function computation and to update the weights in the new sub-network.

In this way, a new network that is capable of classifying new inputs using the support information provided by the original one is obtained. After the retraining phase, the inference of an input vector uses both networks: the final estimation is obtained by averaging the estimations of both models (the original and the new one). This approach has the advantage of avoiding the retrain of the whole system but only a part of it, and of using a limited data set composed of only altered input data. This led to a shorter training time than the one required for the data augmentation technique. Moreover, the original network is not modified; this is an advantage, as modifications are sometimes not possible if the model is read-only or available only as a black box. Furthermore, keeping the original network unaltered is more likely to maintain the same performance on unaltered data.

7.6 Robustness in medical devices

As introduced in Sect. 1.2, the certification process for medical devices and systems must follow a set of standards and activities. However, these standards may be difficult to be applied to AI-based medical software, as their behavior is difficult to be predicted and tested. Despite this, nowadays, many medical devices based on AI algorithms are certified by competent authorities [20], but a lack of clarity on the approval of AI/ML-based medical devices and algorithms characterizes the certification process. From the information available on the already certified AI-based medical software, it can be noticed that the FDA approves AI-based medical systems in three cases:

i) AI algorithms have shown to be at least as safe and effective as another similar legally marketed product (which normally does not rely on AI);

ii) Critical algorithms with high impact on humans are pre-market approved, then the FDA determines if the device's safety and effectiveness are supported by satisfactory scientific evidence;

iii) Novel medical devices which offer adequate safety and effectiveness are approved after performing a risk-based assessment.

Although no robustness assessment is currently formally required, the companies producing medical devices embedding AI components may include the robustness results as part of risk-assessment documentation because they allow evaluating how the model resists to input perturbations.

7.7 Conclusion

In this chapter, a general discussion about the robustness of NNs used in the medical domain, either for classification or estimation, is presented. Previous attempts of evaluating the robustness of NNs are present in the literature, but they mainly focus on adversarial robustness. However, in some domains (such as the medical one), adversarial attacks are unlikely to occur, and, for this reason, other measures should be used in such fields. Thus, the proposed formulas exploit a quality measure (*MAPE* in the case of estimators, or accuracy in the case of classifiers) to evaluate how the network under analysis is able to tolerate input perturbations without degrading its performance. Moreover, in this chapter, I have also presented two different methods suitable for increasing the robustness of a network. They are based on adding data to the training set, or on automatic modifications of the network without requiring any re-design of the network structure. Note that the concepts presented are rather general and can be applied even to other domains.

As explained in Sect. 7.6, robustness is one of the candidate solutions for assessing the quality of PEMS during their certification process. For this reason, in the next chapter, I will present how these concepts are applied to real-world medical case studies, such as a system for classifying medical images coming from breast-cancer exams and a system for estimating the blood pO_2 during medical surgeries.

Chapter 8. Applying robustness computation and improvement to PEMS

In this chapter, the robustness computation presented in Chapter 7 is applied to real medical case studies. In particular, in Chapter 7, the definition of robustness w.r.t. plausible alteration has been given for NN estimators and classifiers. Here, I present the use of the formula and tools previously described in the case of:

- A CNN used as a classifier, for medical images of histological examinations for breast cancer diagnosis;
- An MLP used as an estimator, to monitor the pO_2 level during surgeries.

Both cases can be classified as safety-critical systems and, in particular, they represent (on their own or in part) examples of PEMS. For this reason, their safety and reliability must be proven, and the robustness measure can contribute to increasing the confidence in the correct behavior of the two systems, even in real situations where input may be disturbed by external factors (see Sect. 7.6 for further details).

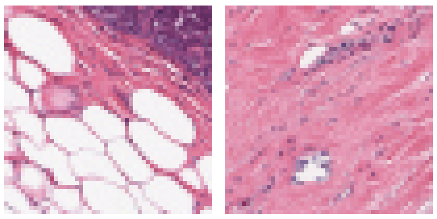
This chapter is based on the work published in [3,5,6] and is structured as follows. Sect. 8.1 presents the description of the breast cancer case study, defines the alterations that have been considered for robustness analysis, presents the robustness results and the improvements obtained with the application of the methods presented in the previous chapter. Sect. 8.2 introduces the pO_2 estimation case study, presents the alterations considered for the MLP estimator, and describes the results of the robustness analysis and improvement processes. Finally, Sect. 8.3 concludes the chapter.

8.1 The breast cancer case study

In this section, I present the application of the robustness computation and improvement process to a Convolutional Neural Network (CNN) used to classify images coming from histological exams for breast cancer diagnosis.

8.1.1 Case study description

Breast cancer (especially invasive ductal carcinoma - IDC) is one of the main causes of cancer death in women (~ 12% in 2019) and one of the most diagnosed cancers (1/3 of all cancers) [47]. The diagnoses for this disease are made by analyzing images of the histological features of tissue or cells removed with surgery or biopsy. These images are collected using a microscope and examined by


(a) Benign examples.

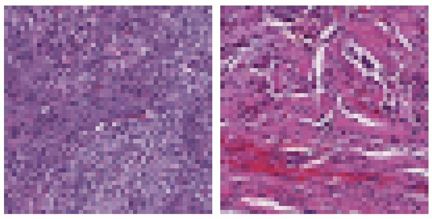

(b) Malignant examples.

Figure 54: Example of images contained in the database used for the analysis.

pathologists to make a decision about the benignity or malignancy of the suspected cancer. For the analyses presented in this section, a publicly available data set of histological images [151] has been used: it consists of 162 images of tissues acquired at 40×, from which a total of $277,524$ labeled patches of 50×50 pixels were extracted. Among them, $198,738$ are benign samples and $78,786$ are malignant. Fig. 54 reports examples of both malignant and benign samples.

For performing this classification, physicians may be aided by automated systems based on CNNs. The CNN analyzed, in the following referred as C_0, is supposed to identify whether the input image comes from a patient with IDC or not. It has been implemented using Python and the Keras library. The structure of the chosen CNN has been inspired by [154] that describes a CNN for breast cancer identification. The first layer in C_0 is a convolutional layer, with 32 filters, and 3×3 kernels, followed by a rectified linear unit (ReLU) activation function. Then, a batch normalization layer, a max-pooling layer, and a dropout of 0.3 are inserted to prevent overfitting. After these layers, a double couple of convolutional layers (64 filters, with 3×3 kernels) and ReLU activation functions are used. To further prevent over-fitting, another batch normalization layer followed by a max-pooling layer is present. The last block of layers is composed of a fully-connected layer with ReLU activation, a batch normalization with a drop-out of 0.5, and a sigmoid classifier.

C_0 has $63,106$ parameters and its training requires $2h\ 08m$. The available $277,524$ input images have been divided as follows: $147,415$ images have been used as training set TR_{C_0}, $63,178$ images have been used as validation set VA_{C_0}, while $66,931$ images compose the test set TE_{C_0}. With this subdivision, C_0 has achieved an accuracy of $86,46\%$ on the test set TE_{C_0}.

Alterations

In order to investigate the robustness of the CNN C_0, the most common alterations that can occur when working with digital images in the medical sector have been considered:

ID	**Alteration**	L_A	U_A	**×n**
GN	Gaussian noise	0	200	40
JC	JPEG Compression	0%	100%	40
VT	Vertical translation	$-4px$	$+4px$	40
HT	Horizontal translation	$-4px$	$+4px$	40
BA	Blur addition	$0px$	$2px$	40
BV	Brightness variation	−50%	+50%	40
ZO	Zoom	100%	200%	40

Table 27: Alteration values used for robustness analysis of breast cancer classification.

- **Horizontal translation** and **Vertical translation**: these alterations may occur when microscopic slides are incorrectly placed, or placed in a way that was not captured by the pictures used to train the network;
- **Brightness variation**: it may occur when different microscopes, having different lamps, are used for image acquisition;
- **Zoom**: images may be acquired using different levels of zoom. In this way, specific features may have different sizes w.r.t. the one seen during the training of the network;
- **Gaussian noise**: it simulates the possible effect of a wrong manipulation of the microscopic slide (e.g., too much dye has been used for contrast) [58]. During the robustness analysis, the variance σ^2 has been altered accordingly with the alteration level;
- **Blur addition**: it may occur due to a small movement of the microscope that acquires the images, causing a loss of focus. For this alteration, a variation of the radius r of the added blur has been considered;
- **JPEG compression**: it may occur when images are transferred in a lossy manner. This alteration has been applied by varying the compression value q.

More details about the lower and upper bound for each alteration (L_A and U_A) and the number of uniformly sampled points for each type of alteration ($\times n$) are reported in Tab. 27.

8.1.2 Robustness evaluation

In this section, I present the results of the robustness evaluation performed with ROBY (previously presented in Sect. 7.4.1) on the CNN C_0 under analysis, using the previously presented alterations, which are already embedded in the tool.

	C_0	C_{DA}		C_{LDA}		C_{IL}		C_{LIL}	
ID	**Rob.**	**Rob.**	Δ	**Rob.**	Δ	**Rob.**	Δ	**Rob.**	Δ
GN	19.5%	100.0%	80.5%	100.0%	80.5%	34.1%	14.6%	34.1%	14.6%
JC	87.8%	97.6%	9.8%	97.6%	9.8%	90.2%	2.4%	90.2%	2.4%
VT	100.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%
HT	100.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%
BA	63.4%	100.0%	36.6%	100.0%	36.6%	100.0%	36.6%	100.0%	36.6%
BV	17.1%	61.0%	43.9%	87.8%	70.7%	17.1%	0.0%	24.4%	7.3%
ZO	100.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%	100.0%	0.0%
AVG	69.7%	94.1%		97.9%		77.3%		78.4%	

Table 28: Robustness for the classifier for breast cancer diagnosis.

To evaluate the robustness of C_0, a threshold $\Theta = 80\%$ has been chosen. Using the uniform robustness formula presented in Def. 18 in Chapter 7, ROBY automatically produces the robustness results as in Tab. 28. They confirm the invariance property of CNNs with respect to geometric transformations [118] (i.e., the classification does not change when some particular geometric transformations are applied): indeed, C_0 has a robustness of 100% in both translations and zoom. In general, the results show that the classifier C_0 was not robust w.r.t. plausible alterations, since its average robustness is 69.7%.

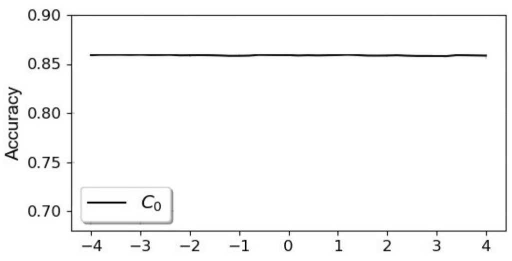
Fig. 55 shows how the accuracy of C_0 varies when each alteration is applied. Apart from the alterations achieving 100% robustness, it can be noticed that some of the other alterations (e.g., JC) maintain the accuracy value greater than Θ for most of their alteration interval, leading to higher robustness values. For other alterations (e.g., GN), instead, the accuracy is lower than Θ for most of the alteration levels, so the robustness is lower.

8.1.3 Robustness improvement

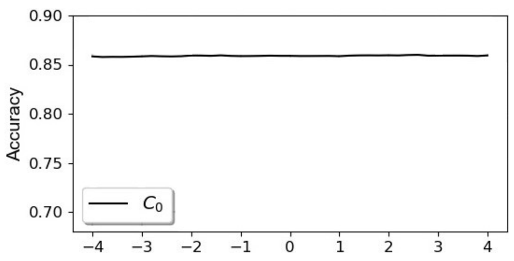
As presented in Sect. 7.5, once the robustness of a NN has been assessed, one may try to improve it by using different techniques, mainly based on adding specific inputs to the training set or adding parallel network components. In this section, the results obtained with the application of the previously presented techniques are reported.

Data augmentation

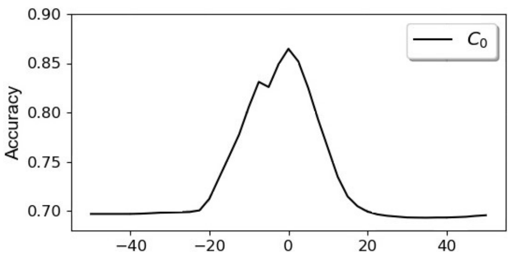
Using *data augmentation* (DA) consists in retraining the entire network using both original and altered pictures. This process is time-consuming, especially if every alteration needs to be applied to each input, for all its alteration levels between L_A and U_A. Based on this consideration, DA has been applied



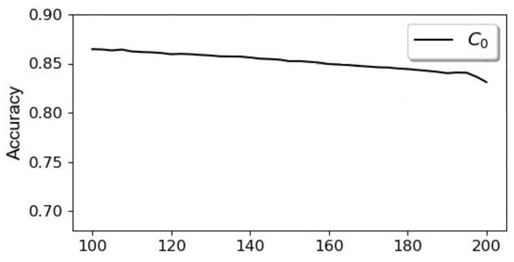
(a) HT: Horizontal Translation (px).



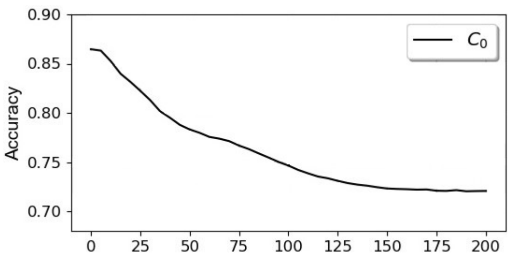
(b) VT: Vertical Translation (px).



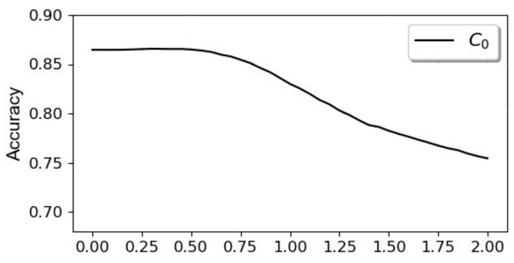
(c) BV: Brightness Variation (%).



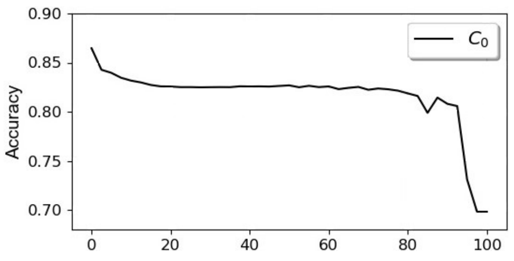
(d) ZO: Zoom (%).



(e) GN: Gaussian Noise (σ^2).



(f) BA: Blur Addition (r).



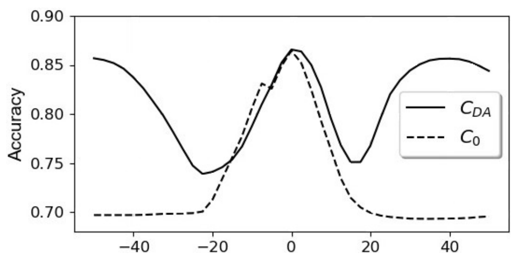
(g) JC: JPEG Compression (q).

Figure 55: Accuracy modification using the altered data input over the CNN C_0.

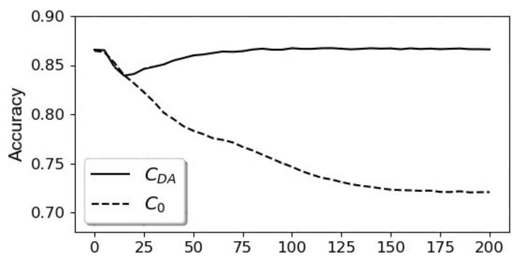
using only 4 alteration levels for the alterations having positive and negative values (VT, HT, and BV) and 2 levels for all the other alterations. In this way, a total of $2,526,281$ input images have been obtained.

The new classifier C_{DA} has required $14h$ $54m$ to be trained and has reached an accuracy of 86.57% on the same test set used for the original classifier C_0. This result shows that the accuracy has not increased significantly but, on the contrary, the robustness w.r.t. alterations has greatly improved (see Tab. 28), both in terms of single alteration (e.g., in the case of Gaussian noise, the robustness has passed from 19.5% to 100.0%) and in terms of average robustness, which now is 94.1%.

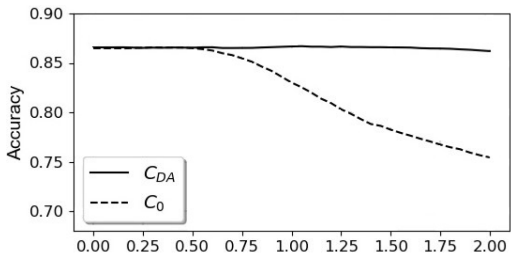
The changes in accuracy obtained by applying the alterations of type A (only for the alterations that lead to a robustness less than 100% with the original CNN C_0) are reported in Fig. 56, as well as the comparison with the original accuracy curves. For most of the alterations, it is possible to observe



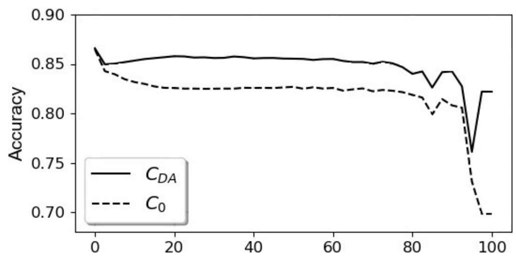
(a) BV: Brightness Variation (%).



(b) GN: Gaussian Noise (σ^2).



(c) BA: Blur Addition (r).



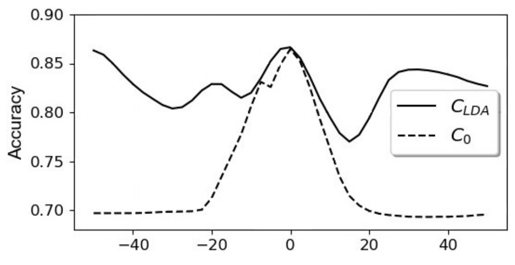
(d) JC: JPEG Compression (q).

Figure 56: Accuracy modification using the altered data input over the CNN C_{DA}.

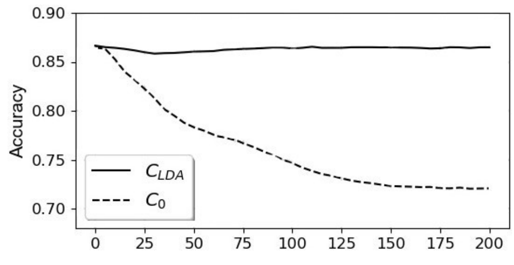
that the accuracy improves more for larger alteration levels than for smaller ones. This may be due to the fact that CNN retraining does not gain enough additional knowledge from small alterations, while it can learn better larger alteration values as they are more distinguishable.

Limited data augmentation

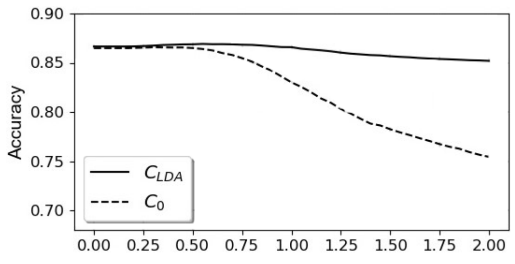
The main disadvantage of the DA technique is that the training process is very time-consuming. Therefore, exploiting the CNNs invariance properties, a "limited" version (LDA) may be used: it is possible to use augmented images only for the alterations that lead to a robustness less than 100% on the original classifier C_0. In particular, the classifier C_{LDA} has been obtained by training the original one with images on which only alterations leading to a robustness less than 100% on C_0: BV, GN, BA, and JC. In this way, 2,316,523 images have been obtained.
The new classifier C_{LDA} has required $13h$ $39m$ to be trained and has reached an accuracy of 86.64% on the same test set used for the original classifier C_0. This result shows that the accuracy has not increased significantly, but, on the contrary, the robustness w.r.t. alterations has greatly improved (see Tab. 28), both in terms of single alteration (e.g., in the case of Gaussian noise, the robustness has passed from 19.5% to 100.0%) and in terms of average robustness, which now is 97.9%. Furthermore, the performance of C_{LDA} is even better than that of C_{DA}: this may be due to the fact that the retraining process focuses only on the weaknesses of the network. The changes in accuracy obtained by applying the alterations of type A (only for the alterations that lead to a robustness less than 100% with the original CNN C_0) are reported in Fig. 56, as well as the comparison with the original accuracy curves.



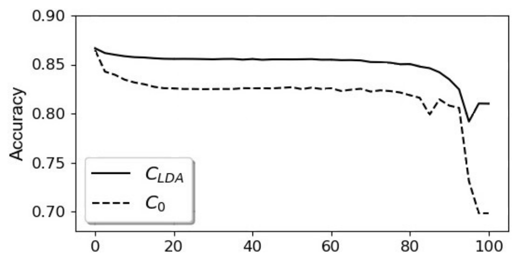
(a) BV: Brightness Variation (%).



(b) GN: Gaussian Noise (σ^2).



(c) BA: Blur Addition (r).



(d) JC: JPEG Compression (q).

Figure 57: Accuracy modification using altered data input over the CNN C_{LDA}.

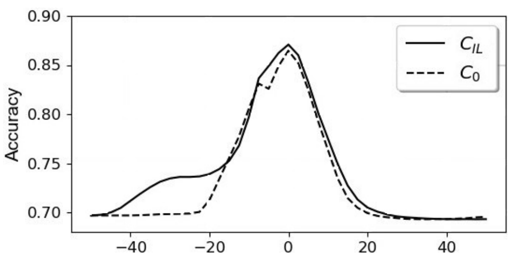
Incremental learning

Using offline *incremental learning* (IL) consists in adding a copy of the original network in parallel to C_0 and training it only using altered input data. In this way, the new network is composed of two branches: one that is more focused on "original" and unaltered data, and one that is able to deal with altered inputs. Then, the result of the entire network $C_{IL} = C_0 || C_{Par}$ is given by the combination (using a max layer) of the outputs of C_0 and C_{Par}.

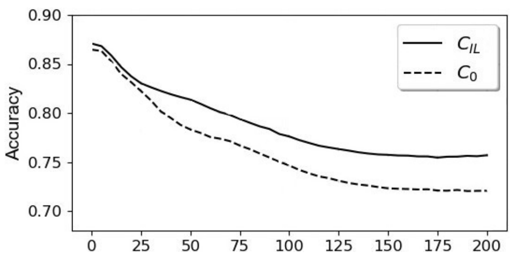
The parallel network retraining has been performed using only the $2,315,688$ images obtained by applying the alterations to the original ones in the input set, and no unaltered data have been used. In particular, as for DA, only 4 alteration levels have been used for alterations with positive and negative values (VT, HT, and BV) and 2 levels for all other alterations. The new classifier C_{IL} has required $13h$ $26m$ to be trained and has reached an accuracy of 87.10% on the same test set used for the original classifier C_0, which is higher than those previously obtained.

The robustness results obtained with C_{IL} are reported in Tab. 28. For all alterations, C_{IL} offers better or equal robustness w.r.t. C_0, but for GN, JC, and BV the techniques based on the classic version of data augmentation (DA or LDA) perform better.

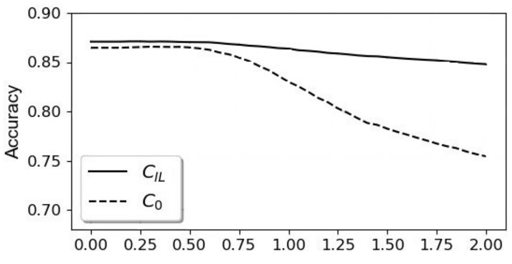
The accuracy changes obtained by applying the alterations of type A (only for the alterations that lead to a robustness less than 100% with the original CNN C_0) are reported in Fig. 58, as well as the comparison with the original accuracy curves.



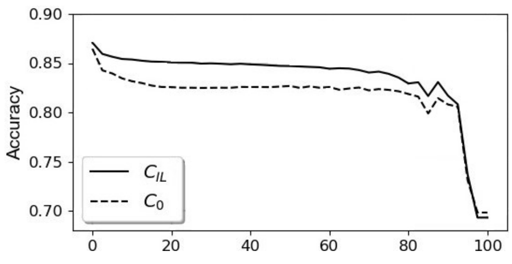
(a) BV: Brightness Variation (%).



(b) GN: Gaussian Noise (σ^2).



(c) BA: Blur Addition (r).



(d) JC: JPEG Compression (q).

Figure 58: Accuracy modification using the altered data input over the CNN C_{IL}.

Limited incremental learning

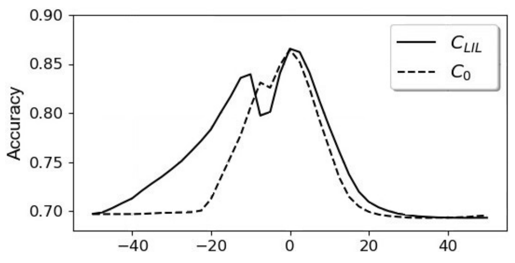
Like the data augmentation technique, incremental learning can be applied in a “limited” version (LIL). For this purpose, the parallel network has been trained using only the $2,105,930$ images obtained by applying the selected alterations (BV, GN, BA, and JC) leading to a robustness lower than 100% on C_0.

The new classifier C_{LIL} has required $12h$ $02m$ to be trained and reached an accuracy of 86.51% on the same test set used for the original classifier C_0, a value lower than that obtained with the full version of incremental learning. However, limiting the input alterations in the training set only to those most critical for the CNN (ad for the LDA technique) allows obtaining better performance than the one achieved with the regular IL technique (see Tab. 28), even if the LDA technique leads to higher robustness in average and for all alterations. The changes in accuracy obtained by applying the alterations of type A (only for the alterations that lead to a robustness less than 100% with the original CNN C_0) are reported in Fig. 59, as well as the comparison with the original accuracy curves.

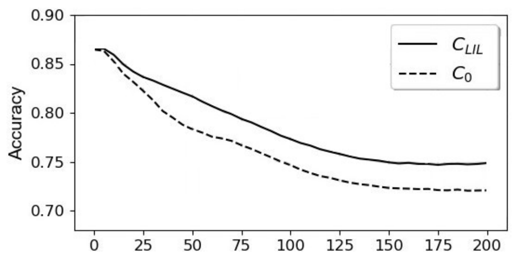
8.1.4 Final considerations

Table 29[1] reports a brief summary of the main relevant information about the four methods presented in the previous subsections. From these results, the best solution is to retrain the whole model using the LDA technique because it leads to a very high resulting average robustness using fewer input images than the standard DA. Nevertheless, in both methods, the training phase requires a lot of time

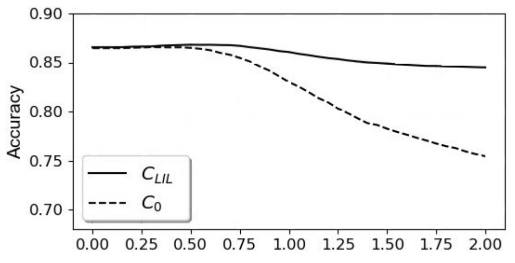
[1]Experiments have been run on a server with 264GB of RAM and a Intel® Xeon® E5-2620 CPU.



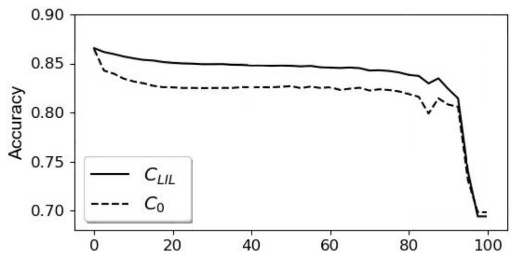
(a) BV: Brightness Variation (%).



(b) GN: Gaussian Noise (σ^2).



(c) BA: Blur Addition (r).



(d) JC: JPEG Compression (q).

Figure 59: Accuracy modification using the altered data input over the CNN C_{LIL}.

C	Input size	Train. time	Accuracy	Avg rob.	Adversariability
C_0	210,593	2h08m	86.46%	69.7%	0.38
C_{DA}	2,526,281	14h54m	86.57%	94.1%	0.64
C_{LDA}	2,316,523	13h39m	86.64%	97.9%	0.39
C_{IL}	2,315,688	13h26m	87.10%	77.3%	N/A
C_{LIL}	2,105,930	12h02m	86.51%	78.4%	N/A

Table 29: Summary of the main information of all the methods used to improve the robustness of a CNN.

(also considering that the data set considered is medium-small). Even the use of the incremental learning technique can lead to an improvement of the robustness of our CNN and, also in this case, the application of the limitation technique can slightly improve the performances in terms of both training time and average robustness.

In general, the networks obtained with all the methods suitable for increasing the robustness seem to perform better than the original one, especially for higher alteration levels. This is reasonable because including an image that is only slightly altered in the input set does not give enough increase of knowledge to the network to significantly improve the accuracy at a certain level of alteration.

Adversariability

Tab. 29 also reports the summary of the adversariability data for the classifiers C for which it is possible to compute it. Note that it is not possible to define "the most adversarial example" as in Def. 20 for networks used with offline incremental learning, since an input that can be adversarial for a part of the net may be not adversarial for the other.

The computed adversariability does not appear to be correlated with robustness: good adversariability values are obtained for both C_0 and C_{LDA} that are very different in robustness. This seems to confirm that the testing based on plausible alterations (aiming at increasing robustness) is complementary to that based on adversarial examples (aiming at reducing adversariability); however, further experiments with other case studies are needed to generalize the results.

8.2 The PO2 estimation case study

In this section, I present the application of the robustness computation and improvement process to a Multilayer Perceptron (MLP) used to estimate the pO_2 level in the blood of patients. The described project has been a collaboration with an industrial partner working in the field of producing medical devices incorporating AI components.

8.2.1 Case study description

In medical practice, constantly evaluating the right value of the partial pressure of oxygen (pO_2) in the blood is very important, especially during surgery or for patients with critical conditions. Normally, the pO_2 level is computed by observing the blood fluorescence. In fact, when exposed to a bright pulse, the blood responses with a fluorescence that can be described (or better "approximated") by a biexponential function defined as follows:

$$fluorescence(t) = A \cdot (e^{-B_1 t} - e^{-B_2 t}) \tag{8.1}$$

where A, B_1, and B_2 are parameters that characterize the response, and t is the time that has passed from the moment in which the light pulse was applied. Note that the parameters A, B_1, and B_2 have been demonstrated to depend on the current level of pO_2 and on blood temperature. An example of this curve (experimentally taken) is shown in the center of Fig. 60. Using a spotlight to illuminate the blood and a probe to measure the response to the bright pulse, it is possible to estimate the parameters A, B_1, and B_2 and then produce an estimation of the pO_2 level. However, finding the best biexponential curve (i.e., finding the fittest parameter values) is very challenging, and it proved to be unfeasible by the microcontroller that the industrial partner had chosen to use in the sensor, since the complexity of the estimation is very high and needed more computational power.

For this reason, the company decided to deploy on the microcontroller an MLP (Multilayer Perceptron), previously trained for the estimation of pO_2. An overview of the MLP-based sensor is shown in Fig. 60.

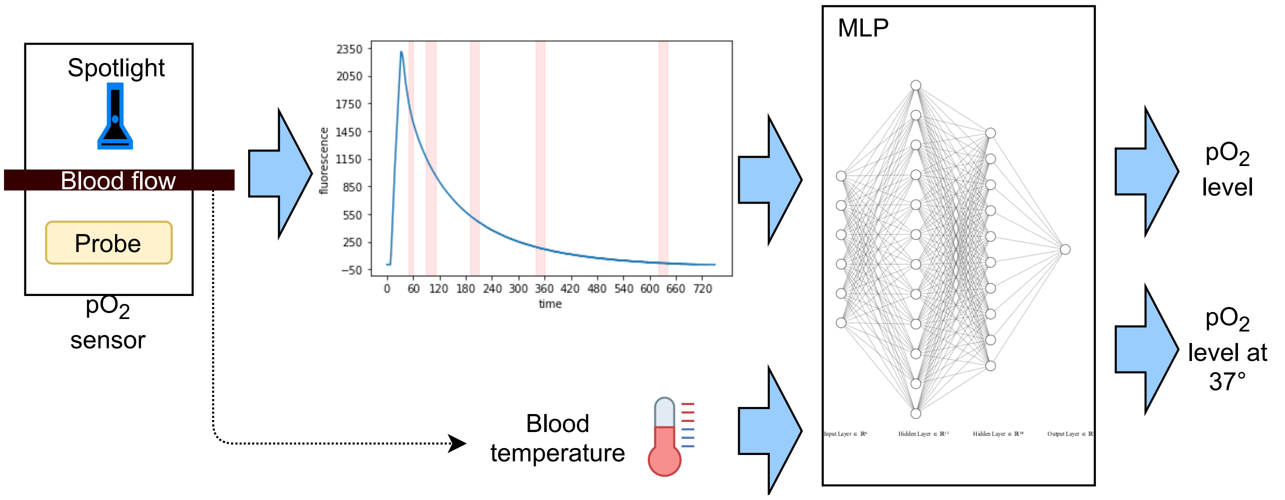


Figure 60: Overview of the MLP-based sensor for pO_2 estimation.

The deployed MLP takes as input a limited number of blood fluorescence samples obtained in response to the bright pulse and the blood temperature to estimate the values of the pO_2 at two temperatures: at the current temperature and at 37 °C. In particular, the MLP uses the mean values of the curve computed in the intervals $[50, 60]$, $[90, 110]$, $[190, 210]$, $[340, 360]$, and $[620, 640]$. The MLP has 6 neurons in the input layer, 12 neurons in the first hidden layer, 10 neurons in the second hidden layer, and 2 neurons in the output layer, giving as output the estimation of the pO_2 value at the current temperature and the prediction of pO_2 value at 37 °C (see Fig. 60). Sigmoid activation functions are used by all neurons. During training, validation, and testing, the company used a data set composed of $21,650$ curves like the one in the center of Fig. 60. The curves have been sampled using 16 different types of probes and 178 different spotlights. For each input sample, the true values of pO_2 at the two temperatures (i.e., current and 37 °C) were given by blood analysis using a precision measurement instrument. As usual in machine learning, 60% of the data set has been used for the training phase, 20% for the validation phase, and 20% for the testing phase.

Alterations

In a real scenario, the data to be processed by the MLP can be altered w.r.t. their nominal shape, as defined by Def. 15. For example, the MLP under analysis has been trained on data that describe how pO_2 changes in time, but in practice, the MLP can be affected in its estimations by variations in acquisition time, due to clock offsets or to a cut of the communication between the sensor and the processing unit. For this reason, in this case study, the industrial partner has asked to consider the following alterations:

- **Cut of the curve end**: it consists in “cutting” the end of the curve obtained in response to the spotlight pulse (see Fig. 61a). This alteration mimics a real situation revealed during testing activities by the industrial partner in which a disruption, failure, or anomalous system behavior leads to a loss of the final part of the curve during its acquisition. From the analyses conducted by the industrial partner, it has been noticed that only cuts in the last range significantly affect the prediction of MLP, i.e., in $[620, 640]$. The alteration is implemented by choosing a t in that range and setting the response to zero after t.

- **Clock offset**: it represents the difference of time calculation in different systems (see Fig. 61b). This may happen when the microprocessor of the acquisition system is not properly configured or there is a delay in signal generation. In the analyses presented in this section, a maximum offset of $30\,ms$ has been used, since it turned out to be a value that guarantees a clock delay greater than the maximum width of the intervals considered as input by the analyzed system.

- **Cut of the peak**: it aims at representing a cut in the peak of the curve, which simulates saturation events (see Fig. 61c). This is a common phenomenon in electronics, where a signal cannot exceed a specific range of values, due to problems in the acquisition chain or source voltage drops. In these cases, high values of the curve are set to a threshold instead of their original values. The signals analyzed for this case study have a maximum amplitude of 4000 RFU (Relative Fluorescence Units). So, to cover only the relevant and possible values, a cut starting from 1300 RFU to 4000 RFU has been applied.

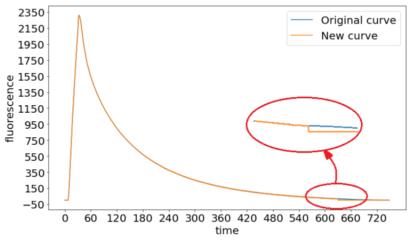
- **Amplification**: it simulates the effect of using different probes and spotlights on the measurement of the same blood sample (see Fig. 61d). In fact, from domain analyses, it has been shown that changing the probes or spotlights (with others made by different manufacturers) slightly amplifies the response curve, even if the real pO_2 value remains the same. For this reason, in the robustness analysis of the MLP, amplifications up to 200% of the original amplitude have been tested.

- **Attenuation**: it represents the opposite of the amplification, i.e., the signal is attenuated by using different probes and/or spotlights (see Fig. 61e). The two alteration types have been evaluated separately since the industrial partner wanted to highlight potential differences between the two. For this alteration, attenuation values up to 50% of the original amplitude have been used.

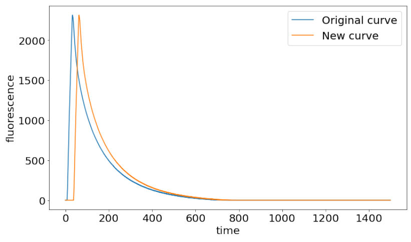
- **Gaussian noise**: it simulates the noises that are common for electronic signals (see Fig. 61f). In this case study, Gaussian noise with a standard deviation in the range between 0 (i.e., the absence

ID	Alteration	L_A	U_A	$\times$ n
CC	Cut of the curve end	620 ms	640 ms	21
CO	Clock offset	0 ms	30 ms	31
CP	Cut of the peak	1300 RFU	4000 RFU	271
AM	Amplification (scale)	100 %	200 %	11
AT	Attenuation (scale)	50 %	100 %	6
GN	Gaussian noise	0	50	51

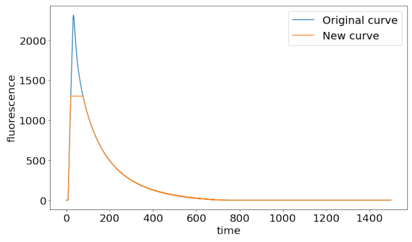
Table 30: Alterations values used for robustness analysis of the pO_2 estimator.



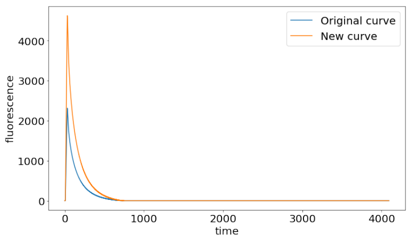
(a) Cut of the curve end.



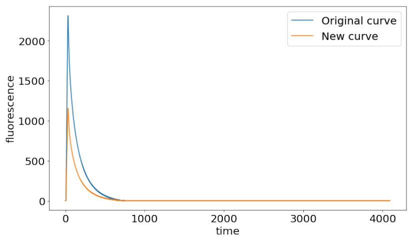
(b) Clock offset.



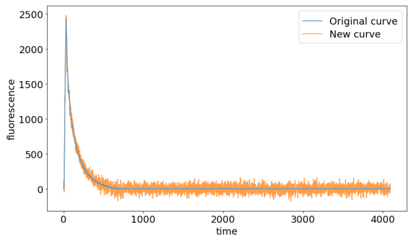
(c) Cut of the peak.



(d) Amplification.



(e) Attenuation.



(f) Gaussian Noise.

Figure 61: Typical alteration examples for the pO_2 estimation case study.

of noise) and 50 (i.e., the maximum value leading to an acceptable and plausible signal-to-noise ratio) has been generated.

More details about the lower and upper bounds for each alteration (L_A and U_A) and the number of uniformly sampled points for each type of alteration ($\times n$) are reported in Tab. 30

8.2.2 Robustness evaluation

In Def. 23 in Chapter 7, the idea of robustness for NN estimators was presented. It is based on the choice of probability for each considered alteration and tolerance for the MAPE. For this reason, the first activity performed was the discussion with the industrial partner about which type of probability and tolerance functions should have been used in this case study. The industrial partner has decided to select *UU robustness*. The choice of uniform probability is motivated by the fact that, in medical applications, sensors should be robust against any alteration level, regardless of the probability of the alteration, since even a very rare alteration level may cause terrible consequences. On the other hand, the choice of uniform tolerance is motivated by the industrial partner, according to physicians

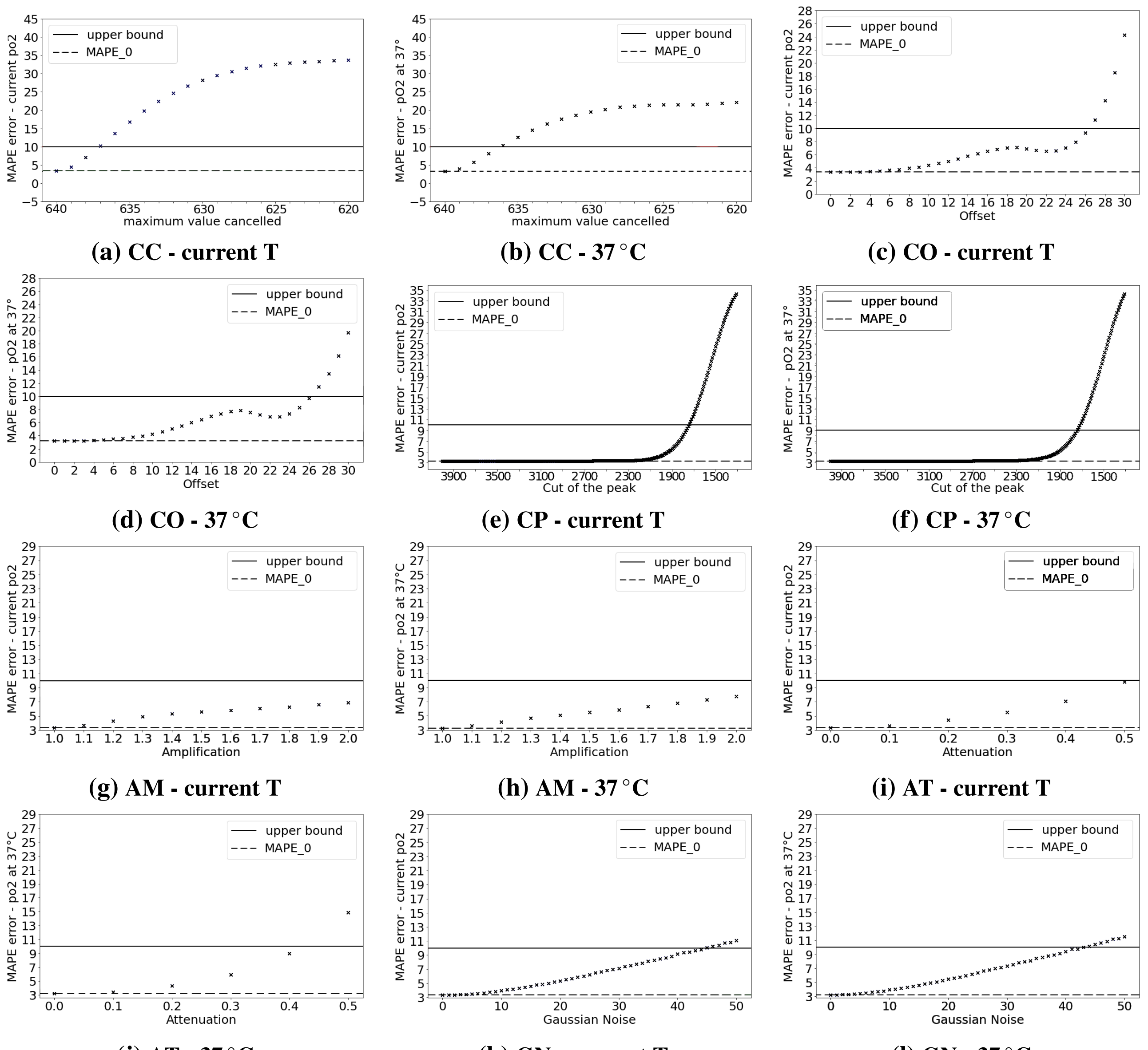
Algorithm 7 Algorithm for robustness analysis.

```
Require: curves_raw, the set of fluorescence curves obtained in response to the bright pulse
Require: targets, the true values of $pO_2$
Require: $M$, the model trained to estimate $pO_2$ values
Require: alteration, the applied alteration
Require: a_levels, the list of $n$ levels uniformly distributed in the range of the chosen alteration, i.e.,
    $[L_A, U_A]$
Require: prob, the probability distribution of the alteration to be applied
Require: Tol, the desired tolerance function
Require: intervals, the list of intervals on which the mean values have to be computed (i.e., $[50, 60]$,
    $[90, 110]$, $[190, 210]$, $[340, 360]$, $[620, 640]$)
Ensure: rob_res, the computed robustness value

    ▷ For each level of alteration
 1: for all $l \in$ a_levels do
    ▷ Apply the alteration level to all the input curves
 2:     alt_curves ← alteration.apply(curves_raw, $l$)
 3:     for all $c \in$ alt_curves do

        ▷ Compute the mean values in the intervals
 4:         meanValues ← compMeanValues($c$, intervals)
            ▷ Compute estimations for altered data
 5:         pred.add($M$.estimate(meanValues))
 6:     end for
        ▷ Compute errors
 7:     MAPE[$l$] = compute_mape(pred, targets)
 8: end for
 9: rob_res ← ROBUSTNESS(MAPE, prob, Tol, a_levels)
10: return rob_res
```

who consider any error below the selected threshold Θ safe for the intended medical practice. More specifically, the industrial partner, after a careful study, has set the maximum accepted MAPE Θ to 10%, a value that physician experts considered safe.

After having decided on the type of robustness to be computed, Algorithm 7 has been executed. Analysis is carried out on a model M, for each *alteration* with uniformly distributed values in an interval *a_levels*, each with a probability described by the probability distribution *prob* (in this case, the uniform function). Moreover, the tolerance function *Tol* (in this case, the uniform function) has to be specified. For each alteration level l (line 1), the algorithm performs the following instructions:

- Starting from *curves_raw*, new altered fluorescence curves are generated (line 2), by applying the defined level l of the *alteration*;
- For each generated curve, the algorithm extracts the mean values in the five intervals of interest (line 4);
- The mean values are used as input for computing the pO_2 estimation (line 5);



(a) CC - current T **(b) CC - 37 °C** **(c) CO - current T**

(d) CO - 37 °C **(e) CP - current T** **(f) CP - 37 °C**

(g) AM - current T **(h) AM - 37 °C** **(i) AT - current T**

(j) AT - 37 °C **(k) GN - current T** **(l) GN - 37 °C**

Figure 62: MAPE variation during robustness analysis.

- The pO_2 results are used to compute the *MAPE* (line 7) for the defined level *l* of the *alteration*;
- After having gathered all the partial *MAPE* values, the function Robustness (line 9) computes the robustness by solving the integral according to Def. 23.

Tab. 31 reports the results obtained by executing Algorithm 7 on the original network. From the results obtained, it is possible to highlight that some alteration is more critical than others. In particular, the alteration for which the robustness of the NN estimator should be improved more is the cut of the curve end (CC). Furthermore, from the comparison between the robustness of the estimation of pO_2 at the current temperature and the one at 37 °C, the results show that there are no relevant differences.

The changes in MAPE obtained when alterations are applied to the test set are reported in Fig. 62, together with the nominal MAPE ($MAPE_0$) and the upper bound Θ used to calculate the robustness.

	Original		**DA-RD**				**DA-AD**				**Incremental learning**			
	Rob [%]		**Rob** [%]		**Δ Rob** [%]		**Rob** [%]		**Δ Rob** [%]		**Rob** [%]		**Δ Rob** [%]	
Alteration	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**	pO_2	pO_2 **37 °C**
CC	14.3	14.3	19.1	19.1	4.8	4.8	28.6	28.6	14.3	14.3	14.3	19.1	0.0	4.8
CO	87.1	83.9	87.1	83.9	0.0	0.0	87.1	87.1	0.0	3.2	87.1	87.1	0.0	3.2
CP	82.9	82.6	83.7	83.7	0.8	1.1	83.3	84.1	0.4	1.5	83.7	83.7	0.8	1.1
AM	81.8	72.7	54.5	54.5	-27.3	-18.2	90.9	72.7	9.1	0.0	100.0	100.0	18.2	27.3
AT	66.7	66.7	66.7	66.7	0.0	0.0	100.0	100.0	33.3	33.3	100.0	83.3	33.3	16.4
GN	86.3	80.4	78.4	84.3	-7.9	3.9	78.4	84.3	-7.9	3.9	88.2	84.3	1.9	3.9
AVG	69.8	66.8	64.9	65.4			78.0	76.1			78.9	76.2		

Table 31: Robustness w.r.t. alterations for the networks retrained with the three approaches.

	# Training	$MAPE_0$ [%]	
Model	**curves**	pO_2	pO_2 **37 °C**
Original	12,990	3.70	3.35
DA-RD	14,677	3.08	3.08
DA-AD	19,485	3.11	3.06
IL	6,495	3.32	3.20

Table 32: $MAPE_0$ **of the original network and the retrained ones.**

8.2.3 Robustness improvement

After having calculated the robustness for the pO_2 estimator, the industrial partner asked to find a way (if existing) to improve the robustness of his MLP model, with only minimal changes in the architecture of the system. As presented in Sect. 7.5, techniques based on data augmentation or incremental learning may be suitable for this purpose. Thus, in this section, three different methods are examined: (A) Data augmentation with recombined data; (B) Data augmentation with altered data; (C) Incremental learning.

Summary data from the analyses are reported in Tab. 31 and Tab. 32, and are discussed in more detail in the following.

Data augmentation with recombined data

As explained previously (see Sect. 7.5.1), the use of data augmentation consists in retraining the entire network using additional data in order to enhance the performance of the NN estimator. In the case study under analysis, it has been observed that there exist many curves that, although they represent the same labeled pO_2 true value, differ in shape, mainly due to differences in temperature and types of probes and/or spotlights. For this reason, the idea at the basis of *data augmentation with recombined data* (DA-RD) is to create new input curves in two different ways: by averaging two curves with out-of-range estimation errors (i.e., higher than 10%), and by averaging a curve with a high estimation

error with one with a low error. In both cases, the curves that have been averaged have the same target values of pO_2. The intent is to capture new intermediate curves with a known true value of pO_2.
With the available data (1141 samples with high estimation error), 546 new data have been generated by the first method and additional 1141 data with the second one, obtaining a new data set composed of 1687 samples that have been added to the original data set and used to retrain the MLP of the industrial partner.
The results in Tab. 31 show that the average robustness obtained when applying DA-RD is lower than the one of the original network, both for the pO_2 at the current temperature and at 37 °C: the robustness w.r.t. the majority of the alterations slightly increases (or remains equal), while for the amplification and Gaussian noise alteration it has significantly decreased. Nevertheless, from the results reported in Tab. 32, it can be seen that this technique has reduced $MAPE_0$, compared to that of the network trained with the original data.
At the end of the day, this has proven not to be a very good solution, but it was not a surprise since, by using this technique, no new data that could mimic possible unexpected alterations were added.

Data augmentation with altered data

The second proposed retraining approach considers the fact that the robustness definition is given w.r.t. plausible alterations. Therefore, unlike the previous approach, data augmentation can be performed with *altered data* (DA-AD), using a retraining approach that explicitly targets alterations and adds altered data during the training phase. However, as usual, data augmentation may take a long time, so only alteration levels that cause a variation of $MAPE$ lower than 5% have been used to create new input data. In this way, a new training data set 1.5 times greater than the original has been obtained.
Tab. 32 shows that the use of DA-AD contributes to the decrease in nominal MAPE, so it improves performance on unaltered data, both for pO_2 at current temperature and at 37 °C. Moreover, results in Tab. 31 confirm that using DA-AD leads to higher average robustness (13% more than the one obtained with the original network) and increases the robustness w.r.t. specific alterations even more than 33% (see AT alteration). Nevertheless, the retraining process requires more time since it is performed using nearly $7,000$ additional input data. Note that the only alteration for which robustness has decreased is Gaussian noise during the estimation of pO_2 at the current temperature. It can be conjectured that this is due to the noise that is, by definition, randomly distributed, and so increasing the robustness w.r.t. this kind of alteration may be very difficult: including in the training set input data subject to Gaussian noise may lead the network to focus only on the specific noise instances, while others may be made more difficult to be tolerated.

Incremental learning

DA, in all its different variants, contributes to improving the robustness performance of a network, but requires retraining of the original network with a larger set of inputs, and this may require a lot of time. Thus, in order to reduce the learning time and increase the network generalization at the same time, incremental learning (IL), as presented in Sect. 7.5.2, has been applied.
This approach has the advantage of avoiding the retraining of the entire system but only a part of it, using a limited data set composed of only altered input data. This led to a shorter training time than the one required for the DA-AD technique, since only the altered data are used to train the new part.
The results in Tab. 31 show that the average robustness obtained when applying IL is the highest, and more than 13% better than the original network. Also, if one wants to consider specific alterations, the robustness has increased up to 33%, and no decrease has been obtained (differently from DA-AD, for which some decrease has been observed). Note that $MAPE_0$, instead, is slightly higher than the one obtained with DA-AD (see Tab. 32), but still better than the original network one.
These results show that IL can combine the advantages of the original network, i.e., focusing only on relevant input features, and those of the data augmentation, i.e., guaranteeing higher robustness w.r.t. the standard-trained network.

8.2.4 Final considerations

In the solution proposed to the industrial partner, the user needs to select the probability of the alterations of interest. Although the probabilities in this case study were known, in some different application scenarios, they may be unknown, and this can make it difficult to apply the formula for robustness computation. In these cases, interpreting probability as a function that describes the "importance" (or weight) of each alteration level is a viable solution.
From the experiments with the three robustness improvement techniques (see Sect. 8.2.3), it can be noticed that, for most alterations, the robustness is not 100% even after the robustness improvement process. This shows that obtaining optimal robustness by simply retraining may not be possible. In this case, by looking at the final robustness results, the industrial partner has understood which are the most critical alterations that can still affect the network and has planned to adopt countermeasures from an electronic point of view. For example, they have experimented that the impact of Gaussian noise may be reduced by improving the cables' shielding or by amplifying the acquired signal so that the noise is less relevant.
Finally, the industrial partner has detected a significant improvement in the performance of the NN estimator after the described process and, among all the techniques presented, has decided to use

incremental learning, as it increases robustness and decreases the nominal MAPE (i.e., $MAPE_0$), while saving time w.r.t. regular data augmentation.

8.3 Conclusion

In this chapter, the robustness estimation process for both a medical image classifier and a pO_2 estimator has been presented. This process exploits the formulas introduced in Chapter 7 and applies them to real medical case studies, where domain-specific alterations are more likely to occur than the most investigated adversarial examples.

In both case studies, robustness computation has allowed improving the performance of the NN, for both unaltered and altered inputs. In breast cancer classification, after having applied the robustness analysis and computation, robustness and nominal accuracy increased. The same results can be observed in the case of the pO_2 estimator: applying methods to increase the robustness has allowed the industrial partner not only to improve robustness itself in the presence of alterations but also to decrease $MAPE_0$ under nominal and unaltered conditions.

In conclusion, evaluating and improving robustness has proven to be a good option when it comes to increasing the quality and reliability of a NN, which is useful not only when dealing with altered and unforeseeable inputs, but also with "regular" data. This has been confirmed by the industrial partner, which now includes this evaluation process in its current pipeline when developing ML-based solutions.

Conclusions and Future Work

In this book, the problem of developing safe and reliable medical software and systems has been tackled. In particular, starting from the state of the art, empirical guidelines have been presented. They are derived from the experience acquired in the field during the development of a real medical device, namely the MVM ventilator, during the first wave of COVID-19 in Italy. These guidelines propose a software development process based on a mixture between the classical V-model and more recent agile techniques.

During the course of the proposed development process, producing documentation and performing V&V activities is paramountly important, as it is required by the device certification standards, but also because it promotes the quality of the products. In this book, I have presented a method, based on a formal and mathematical notation, that exploits the ASMETA framework to represent the specifications of the system. After having verified and validated the abstract specifications, users can exploit them for generating correct-by-construction code. In this book, this technique has been applied to the MVM case study and to a prototype of a pill box produced by a local company. Moreover, if the code of the system is already available, abstract specifications can be used for performing model-based testing on the actual system: test cases are generated starting from the system's model, together with their oracle, and applied to the real system. This technique has been applied to the MVM case study and the IEEE PHD protocol [102] and has led to an important increase in code coverage and to the discovery of bugs or conformance faults.

If model-based testing is adopted, testers may not have the time to perform exhaustive tests (which, sometimes, is not feasible). For this reason, guidance is needed on how to select test cases. In this book, I have presented how testers may apply Combinatorial Interaction Testing in order to reduce the number of test cases to be performed without losing in fault detection capability. Moreover, to optimize the test generation process, two tools for combinatorial test generation have been presented. Experiments reported in this book on the MVM case study show that, by using them, the time for test suite generators can be reduced by more than 97%. This is a positive result, since if fewer time is needed for test generation, more time can be spent on testing execution, and more bugs are likely to be discovered.

However, model-based testing or code generation from ASM models may not always be applicable, especially when medical devices embed AI-based components. However, there is no well-established method for assessing the quality of this kind of medical device. Even regulations do not succeed in

indicating a way to ensure their quality and reliability. In this book, I have proposed the computation of robustness for neural networks. It evaluates how a NN-based system can resist to input perturbation without changing (under a reasonable limit) its behavior. This approach has been tested in two different real systems, i.e., a CNN used for breast cancer diagnosis and an MLP used for the estimation of the blood pO_2 during surgeries. In both cases, the definition of robustness has been shown to be applicable, and the methods devised to enhance the robustness performance of the NNs have led to improved results.
Note that all methodologies and techniques presented in this book have been devised to comply with the main standards available for this field, namely IEC 62304 [64] and the FDA Guidelines [83].

In Chapter , I reported five different research questions that have guided the work presented and that have been addressed in this book. In the following, some final considerations and remarks are made about each of them.

- **RQ1: State of the art in software quality -** In Chapter 1, I have presented the concept of software quality, which is a broader concept and is not limited only to medical devices. In practice, software systems are considered to be of good quality if they comply with their specifications. When it comes to medical systems, software quality assurance is subject to several steps that developers and system producers must follow throughout the life cycle. In this case, international certification standards and regulations (e.g., IEC 62304 [64] and the FDA general principles [83]) must be fulfilled. In addition, other standards for the specific medical device should be analyzed.

- **RQ2: Guidelines for the development process -** In Chapter 3, I have presented the experience gained during the development of the MVM ventilator. Through this experience, several lessons learned and guidelines, to be considered especially when working on a safety-critical system under emergency, have been outlined. The major contribution of this chapter is the software development process, which includes both the classical V-Model and agile aspects, in order to combine the flexibility given by agile (useful to reduce the time-to-market) with the documentation required both in the V-model and for medical software certification. Moreover, other aspects are addressed by the proposed guidelines, e.g., the importance of testing, the need of a coordination team, the gain structuring the software in different modules, with the isolation of safety-critical components, etc.

- **RQ3: Using ASMs for the development of medical systems -** In Chapters 4 and 5, I have presented, respectively, the ASMETA framework and its application to real medical case studies, such as MVM, the e-Pix pill box, and the PHD protocol. The use of ASMETA has allowed the research team to write formal specifications of the systems under analysis, verify and validate them, and, finally, to obtain executable source code or unit test cases. Note that ASMETA has been shown to comply with current regulations on medical software certification (see Sect. 5.1) and to allow users to fulfill to the majority of the activities required. Moreover, the software quality (i.e., the compliance with the system specifications) is ensured when deriving source code from ASM specification, since the verified properties at the model level are maintained even at code level. Thus, the ASMETA framework allows for obtaining a correct-by-construction code. Even using ASMETA for model-based testing has shown to be effective, since it avoids errors that human beings can make while writing tests manually and has allowed the discovery of several bugs or conformance faults on both the MVM and PHD protocol.

- **RQ4: Using CIT for testing medical software -** In Chapter 6, I have presented the Combinatorial Interaction Testing technique and two tools for generating test suites achieving the intended combinatorial coverage. This technique has been proven to be optimal when exhaustive testing cannot be performed, especially for systems with a high number of inputs and outputs, such as medical systems. In fact, CIT allows testers to reduce the number of tests to be performed without reducing the fault detection capability. However, generating this kind of test suites may be time-expensive, so in this book two tools exploiting multi-threading have been proposed. Through experiments on the MVM case study, the former, pMEDICI, has been shown to be the best performing, especially in terms of generation time. However, it cannot deal with relational and other types of complex constraint, so the KALI tool has been introduced.

- **RQ5: How to validate AI-based medical software -** In Chapters 7 and 8, I have presented, respectively, the robustness measure for neural networks and its application to two different medical case studies, i.e., a CNN used for breast cancer diagnosis and an MLP used for the estimation of blood pO_2 during surgeries. Despite international standards and regulations on medical software certification being not yet updated for AI-based medical devices, certification authorities certify medical software relying on AI components under certain guarantees that they perform as good as other devices which do not employ AI. In addition, a risk-assessment activity is required for them. The proposed robustness measure can be used in the context of risk assessment, as confirmed by the industrial partner that has participated in the research activity on MLP for the estimation of blood pO_2. Furthermore, measuring the robustness has been proven to be effective not only for evaluating the performance of a NN, but also to guide its enhancement through the proposed methodology (based on data augmentation or incremental learning).

Future work

The work presented in this book can be further extended in several directions. For this reason, in the following, I report some of the future work that can be investigated and motivate why they are important for increasing the process of quality assurance of medical devices:

- During the development of the MVM ventilator (see Chapter 2), one of the main problems has been the difficulty in testing software that needs to interact with humans and the external environment without actually having them. This limitation has been highlighted in this book in Chapter 3 and a guideline on it has been reported in Sect. 3.4.2. Since MVM has proved to be an important and exhaustive case study, it could be used for further investigation on several techniques. However, a complete and configurable *digital twin* [165] should be developed to make MVM software more easily developed and tested. Note that the usefulness of a digital twin is also motivated by the work of the community of `Models-at-runtime`, in which this kind of twin is also used during the operation of the device. Indeed, it can be exploited not only for device testing but also to be executed together with the actual system to check its correct behavior and to forecast possible future critical conditions.

- Additional work may be done on the MVM ventilator (see Chapter 2) in order to increase its usability. In particular, some of the other ventilators on the market implement an adaptive ventilation strategy which is used for defining automatically the ventilation parameters based on patient's condition. This will allow to reduce the possible errors made by physicians and to increase the "ventilation comfort" for patients subject to mechanical ventilation.

- In Chapters 4 and 5, I have shown how ASMETA (and, in general, the ASM formalism) can be applied to medical systems to satisfy most of the requirements of certification standards. However, there is still an important aspect that has not yet been implemented and managed by the ASMETA framework. In particular, a common problem in the development of medical software is that the uncertainties of the system and its environment should be considered [75]. For example, a patient under mechanical ventilation may change his condition with a known or unknown probability, the backup battery of a medical system may fail with a known probability, etc. Thus, ASMETA should be extended with probabilistic features in order to model and develop systems under some uncertainties.

- In Chapter 7, I have presented the novel concept of robustness that can be used when dealing with medical software embedding NNs. It has been tested and approved by an industrial partner on a real system and has been shown to be effective in evaluating the quality of the estimations or predictions made by a NN. However, in medical systems, not all errors are the same: telling a patient that he is affected by a disease when it is not true is not the same as telling him that he is healthy while he is not. For this reason, additional measures should be considered in order to weight more errors that are more critical.
- In Chapter 7, I have presented a way to speed up the robustness computation (see Sect. 7.4.2), which is based on the parabolic approximation of the accuracy curve. However, the proposed method has been shown to work well under the assumption that the parameter $\hat{a}$ is chosen coherently with the system to be analyzed, but there is no guide available on how to choose its value. For this reason, as future work, heuristics or more detailed guidance should be defined in order to make the ASAP approach more usable in practice.

List of Figures

List of Tables

List of Listings

References

[1] EU 2017/745 Medical devices - Regulation (EU) 2017/745 of the European Parliament and of the Council of 5 April 2017 on medical devices, amending Directive 2001/83/EC, Regulation (EC) No 178/2002 and Regulation (EC) No 1223/2009 and repealing Council Directives 90/385/EEC and 93/42/EEC.

[2] A. Abba et al. The novel mechanical ventilator milano for the COVID-19 pandemic. *Physics of Fluids*, 33(3):037122, mar 2021.

[3] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. Dealing with robustness of convolutional neural networks for image classification. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 7–14, 2020.

[4] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. Efficient computation of robustness of convolutional neural networks. In *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, aug 2021.

[5] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. ROBY: a tool for robustness analysis of neural network classifiers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2021.

[6] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Daniele Gamba, and Rita Pedercini. Robustness assessment and improvement of a neural network for blood oxygen pressure estimation. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322, 2022.

[7] Paolo Arcaini, Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. The ASMETA approach to safety assurance of software systems. In *Logic, Computation and Rigorous Methods*, pages 215–238. Springer International Publishing, 2021.

[8] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor, and Elvinia Riccobene. Integrating formal methods into medical software development: The ASM approach. *Science of Computer Programming*, 158:148–167, jul 2018.

[9] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Visual notation and patterns for Abstract State Machines. In Paolo Milazzo, Dániel Varró, and Manuel Wimmer,

editors, *Software Technologies: Applications and Foundations*, pages 163–178, Cham, 2016. Springer International Publishing.

[10] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the Second International Conference on Abstract State Machines, Alloy, B and Z*, ABZ'10, pages 61–74, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic review of Abstract State Machines by meta property verification. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[12] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, Lecture Notes in Computer Science, pages 253–269, Cham, 2016. Springer International Publishing.

[13] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.

[14] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41:155–166, 2011.

[15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Trans. Auton. Adapt. Syst.*, 11(4):25:1–25:35, January 2017.

[16] DavidG. Ashbaugh, D. Boyd Bigelow, ThomasL. Petty, and BernardE. Levine. Acute respiratory distress in adults. *The Lancet*, 290(7511):319 – 323, 1967. Originally published as Volume 2, Issue 7511.

[17] ASMETA (ASM mETAmodeling) toolset. `https://asmeta.github.io/`.

[18] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[19] Sigall K. Bell, Tom Delbanco, Joann G. Elmore, Patricia S. Fitzgerald, Alan Fossa, Kendall Harcourt, Suzanne G. Leveille, Thomas H. Payne, Rebecca A. Stametz, Jan Walker, and Catherine M. DesRoches. Frequency and types of patient-reported errors in electronic health record ambulatory care notes. *JAMA Network Open*, 3(6):e205867, June 2020.

[20] Stan Benjamens, Pranavsingh Dhunnoo, and Bertalan Meskó. The state of artificial intelligence-based FDA-approved medical devices and algorithms: an online database. *npj Digital Medicine*, 3(1), sep 2020.

[21] B. Boehm and R. Turner. Balancing agility and discipline: evaluating and integrating agile and plan-driven methods. In *Proceedings. 26th International Conference on Software Engineering*, 2004.

[22] Andrea Bombarda, Silvia Bonfanti, Cristiano Galbiati, Angelo Gargantini, Patrizio Pelliccione, and Elvinia Riccobene. Lessons learned from the development of a mechanical ventilator for COVID-19. In *32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, aug 2021.

[23] Andrea Bombarda, Silvia Bonfanti, Cristiano Galbiati, Angelo Gargantini, Patrizio Pelliccione, and Elvinia Riccobene. Replication package. available online at: `https://se4med.github.io/GuidelinesForCriticalSoftware/`, 2022.

[24] Andrea Bombarda, Silvia Bonfanti, Cristiano Galbiati, Angelo Gargantini, Patrizio Pelliccione, Elvinia Riccobene, and Masayuki Wada. Guidelines for the development of a critical software under emergency. *Information and Software Technology*, page 107061, September 2022.

[25] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. Developing medical devices from abstract state machines to embedded systems: A smart pill box case study. In *Software Technology: Methods and Tools*, pages 89–103. Springer International Publishing, 2019.

[26] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. *Automatic Test Generation with ASMETA for the Mechanical Ventilator Milano Controller*, page 65–72. Springer International Publishing, 2022.

[27] Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Yu Lei, and Feng Duan. RATE: A model-based testing approach that combines model refinement and test execution. *Software Testing, Verification and Reliability*, 33(2), December 2022.

[28] Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, Marco Radavelli, Feng Duan, and Yu Lei. Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using abstract state machines. In *Testing Software and Systems*, pages 67–85. Springer International Publishing, 2019.

[29] Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Developing a prototype of a mechanical ventilator controller from requirements to code with asmeta. *Electronic Proceedings in Theoretical Computer Science*, 349:13–29, Nov 2021.

[30] Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini, and Elvinia Riccobene. Extending ASMETA with time features. In *Rigorous State-Based Methods*, pages 105–111. Springer International Publishing, 2021.

[31] Andrea Bombarda, Edoardo Crippa, and Angelo Gargantini. An environment for benchmarking combinatorial test suite generators. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, April 2021.

[32] Andrea Bombarda and Angelo Gargantini. An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, October 2020.

[33] Andrea Bombarda and Angelo Gargantini. Parallel test generation for combinatorial models based on multivalued decision diagrams. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 74–81, 2022.

[34] Andrea Bombarda and Angelo Gargantini. Incremental generation of combinatorial test suites starting from existing seed tests. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, April 2023.

[35] Andrea Bombarda, Angelo Gargantini, and Andrea Calvagna. Multi-thread combinatorial test generation with smt solvers. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23. ACM, March 2023.

[36] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. AsmetaA: Animator for Abstract State Machines. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors,

Abstract State Machines, Alloy, B, TLA, VDM, and Z, pages 369–373, Cham, 2018. Springer International Publishing.

[37] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Generation of C++ unit tests from Abstract State Machines specifications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 185–193, April 2018.

[38] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. Design and validation of a C++ code generator from Abstract State Machines specifications. *Journal of Software: Evolution and Process*, 32(2):e2205, 2020. e2205 smr.2205.

[39] Silvia Bonfanti, Elvinia Riccobene, and Patrizia Scandurra. A runtime safety enforcement approach by monitoring and adaptation. In *Software Architecture*, pages 20–36. Springer International Publishing, 2021.

[40] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *2008 IEEE Int. Conf. on Software Maintenance*, 2008.

[41] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. CNN-cert: An efficient framework for certifying robustness of convolutional neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:3240–3247, jul 2019.

[42] Egon Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[43] Egon Börger and Alexander Raschke. *Modeling Companion for Software Practitioners*. Springer, Berlin, Heidelberg, 2018.

[44] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[45] Cari Borrás. Overexposure of radiation therapy patients in panama: problem recognition and follow-up measures. *Revista Panamericana de Salud Pública*, 20(2-3), September 2006.

[46] Michela Botta, Anissa M Tsonas, Janesh Pillay, Leonoor S Boers, Anna Geke Algera, Lieuwe D J Bos, et al. Ventilation management and clinical outcomes in invasively ventilated patients with covid-19 (provent-covid): a national, multicentre, observational cohort study. *The Lancet Respiratory Medicine*, 2020.

[47] Breastcancer.org. U.S. breast cancer statistics. `https://www.breastcancer.org/symptoms/understand_bc/statistics`, 2019.

[48] Laurent J. Brochard and Francois Lellouche. *Chapter 8. Pressure-Support Ventilation*, chapter 8. The McGraw-Hill Companies, New York, NY, 2013.

[49] Frederick P. Brooks. *The Mythical Man Month.* Prentice Hall, 1995.

[50] Roy G Brower, Michael A Matthay, Alan Morris, David Schoenfeld, B Taylor Thompson, and Arthur Wheeler. Ventilation with lower tidal volumes as compared with traditional tidal volumes for acute lung injury and the acute respiratory distress syndrome. *New England Journal of Medicine*, 342(18):1301–1308, 2000. PMID: 10793162.

[51] Marie T. Brown and Jennifer K. Bussell. Medication adherence: WHO cares? *Mayo Clinic Proceedings*, 86(4):304–314, apr 2011.

[52] Dominique Brunet, Edward R. Vrscay, and Zhou Wang. On the mathematical properties of the structural similarity index. *IEEE Transactions on Image Processing*, 21(4):1488–1499, 2012.

[53] Rossana Bussani, Edoardo Schneider, Lorena Zentilin, Chiara Collesi, Hashim Ali, Luca Braga, Maria Concetta Volpe, Andrea Colliva, Fabrizio Zanconati, Giorgio Berlot, Furio Silvestri, Serena Zacchigna, and Mauro Giacca. Persistence of viral rna, pneumocyte syncytia and thrombosis are hallmarks of advanced covid-19 pathology. *EBioMedicine*, page 103104, 2020.

[54] DONALD CAMPBELL and JAMES BROWN. The electrical analogue of lung. *British Journal of Anaesthesia*, 35(11):684–692, 1963.

[55] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for ASMs. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, ABZ '08, pages 71–84, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.

[57] B. Chacko, J.V. Peter, P. Tharyan, G. John, and L. Jeyaseelan. Pressure-controlled versus volume-controlled ventilation for acute respiratory failure due to acute lung injury (ali) or acute respiratory distress syndrome (ards). *Cochrane Database of Systematic Reviews*, (1), 2015.

[58] Shailja Chatterjee. Artefacts in histopathology. *Journal of Oral and Maxillofacial Pathology*, 18(4):111, 2014.

[59] Michel R. V. Chaudron, Werner Heijstek, and Ariadi Nugroho. How effective is UML modeling ? *Software & Systems Modeling*, 11(4), 2012.

[60] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 248–254, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[61] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, Berlin, Heidelberg, 2002. Springer-Verlag.

[62] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[63] International Electrotechnical Commission. IEC 60601-1:2005 Medical electrical equipment - Part 1: General requirements for basic safety and essential performance. Technical report, International Electrotechnical Commission, 2005.

[64] International Electrotechnical Commission. IEC 62304 Medical device software — Software life cycle processes. Technical report, International Electrotechnical Commission, 2006.

[65] International Electrotechnical Commission. IEC 61508-3:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements. Technical report, International Electrotechnical Commission, 2010.

[66] International Electrotechnical Commission. IEC 61784 - Industrial communication networks. Technical report, International Electrotechnical Commission, 2021.

[67] Santosh K. Das, Jan e Alam, Salvatore Plumari, and Vincenzo Greco. Transmission of airborne virus through sneezed and coughed droplets. *Physics of Fluids*, 32(9):097102, September 2020.

[68] Talib Dbouk and Dimitris Drikakis. On coughing and airborne droplet transmission to humans. *Physics of Fluids*, 32(5):053310, May 2020.

[69] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[70] Amanda Dexter, Neil McNinch, Destiny Kaznoch, and Teresa A. Volsko. Validating lung models using the ASL 5000 breathing simulator. *Simulation in Healthcare: The Journal of the Society for Simulation in Healthcare*, 13(2):117–123, apr 2018.

[71] Florian Dubost, Gerda Bortsova, Hieab Adams, M. Arfan Ikram, Wiro Niessen, Meike Vernooij, and Marleen de Bruijne. Hydranet: Data augmentation for regression neural networks. In *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019: 22nd International Conference, Shenzhen, China, October 13–17, 2019, Proceedings, Part IV*, page 438–446, Berlin, Heidelberg, 2019. Springer-Verlag.

[72] Christian Dufour, Guillaume Dumur, Jean-Nicolas Paquin, and Jean Belanger. A pc-based hardware-in-the-loop simulator for the integration testing of modern train and ship propulsion systems. In *2008 IEEE Power Electr. Specialists Conf.*, 2008.

[73] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[74] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007.

[75] Prashanti Eachempati, Roland Brian Büchter, Kiran Kumar KS, Sally Hanks, John Martin, and Mona Nasser. Developing an integrated multilevel model of uncertainty in health care: a qualitative systematic review and thematic synthesis. *BMJ Global Health*, 7(5), 2022.

[76] Jutta Eckstein. Architecture in large scale agile development. In *Lecture Notes in Business Information Processing*. 2014.

[77] Wasim Essbai, Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. A framework for including uncertainty in robustness evaluation of bayesian neural network classifiers. In *2024 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, May 2024.

[78] Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. Robustness of classifiers: from adversarial to random noise. In *Advances in Neural Information Processing Systems 29*, pages 1632–1640. 2016.

[79] Michael Felderer and Rudolf Ramler. Risk orientation in software testing processes of small and medium enterprises: an exploratory and comparative study. *Software Quality Journal*, 24(3), August 2015.

[80] Stanley Feldman. The Manley ventilator. *Anaesthesia*, 50(1):64–71, January 1995.

[81] Carlos Ferrando, Fernando Suarez-Sipmann, Ricard Mellado-Artigas, María Hernández, Alfredo Gea, Egoitz Arruti, et al. Clinical features, ventilatory management, and outcome of ARDS caused by COVID-19 are similar to other causes of ARDS. *Intensive Care Medicine*, 46(12):2200–2211, July 2020.

[82] Gabriel Ferreira, Christian Kästner, Joshua Sunshine, Sven Apel, and William L. Scherlis. Design dimensions for software certification: A grounded analysis. *CoRR*, abs/1905.09760, 2019.

[83] Food and Drug Administration. *General Principles of software validation; final guidance for industry and FDA staff, version 2.0.* U.S Food and Drug Administration (FDA), Jan. 2002.

[84] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: A survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, September 2009.

[85] A. Gannous and A. Andrews. Integrating safety certification into model-based testing of safety-critical systems. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–260, 2019.

[86] Sergio Garcia, Daniel Struber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. An empirical assessment of robotics software engineering. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[87] A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, April 2018.

[88] Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *J. Univers. Comput. Sci.*, 7(11):1050–1067, 2001.

[89] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to generate tests from ASM specifications. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, pages 263–277, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[90] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Ten reasons to metamodel ASMs. In Jean-Raymond Abrial and Uwe Glässer, editors, *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, pages 33–49, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[91] Angelo Gargantini and Paolo Vavassori. Efficient combinatorial test generation based on multivalued decision diagrams. In Eran Yahav, editor, *Hardware and Software: Verification and Testing*, pages 220–235, Cham, 2014. Springer International Publishing.

[92] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *2009 1st International Symposium on Search Based Software Engineering*, pages 13–22, 2009.

[93] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, jul 2010.

[94] L Gattinoni and A Pesenti. The concept of “baby lung”. *INTENSIVE CARE MEDICINE*, 31(6):776–784, JUN 2005.

[95] Domenico Luca Grieco, Filippo Bongiovanni, Lu Chen, Luca S. Menga, Salvatore Lucio Cutuli, Gabriele Pintaudi, Simone Carelli, Teresa Michi, Flava Torrini, Gianmarco Lombardi, Gian Marco Anzellotti, Gennaro De Pascale, Andrea Urbani, Maria Grazia Bocci, Eloisa S. Tanzarella, Giuseppe Bello, Antonio M. Dell’Anna, Salvatore M. Maggiore, Laurent Brochard, and Massimo Antonelli. Respiratory physiology of COVID-19-induced respiratory failure compared to ARDS of other etiologies. *Critical Care*, 24(1), AUG 28 2020.

[96] Yuri Gurevich. *Evolving Algebras 1993: Lipari Guide*, pages 9–36. Oxford University Press, Inc., USA, 1995.

[97] Christopher Henard, Mike Papadakis, and Yves Le Traon. Flattening or not of the combinatorial interaction testing models? In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4. IEEE, apr 2015.

[98] Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679–688, 2006.

[99] Claire Ingram and Steve Riddle. Cost-benefits of traceability. In *Software and Systems Traceability*, pages 23–42. Springer London, London, 2012.

[100] ISO Central Secretary. UNI CEI EN ISO 16000-6:2011 Indoor air — Part 6: Determination of volatile organic compounds in indoor and test chamber air by active sampling on Tenax TA sorbent, thermal desorption and gas chromatography using MS or MS-FID. Technical report, International Organization for Standardization, 2011.

[101] ISO Central Secretary. UNI EN ISO 9001:2015 Quality management systems - Requirements. Technical report, International Organization for Standardization, 2015.

[102] ISO Central Secretary. ISO/IEC/IEEE international standard - health informatics – personal health device communication - part 20601: Application profile - optimized exchange protocol. Technical report, International Organization for Standardization, 2016.

[103] ISO Central Secretary. UNI CEI EN ISO 18562-3:2017 Biocompatibility evaluation of breathing gas pathways in healthcare applications — Part 3: Tests for emissions of volatile organic compounds (VOCs). Technical report, International Organization for Standardization, 2017.

[104] ISO Central Secretary. UNI CEI EN ISO 14971:2020 Medical devices - Application of risk management to medical devices. Technical report, International Organization for Standardization, 2020.

[105] ISO Central Secretary. UNI CEI EN ISO 13485:2021 Medical devices - Quality management systems - Requirements for regulatory purposes. Technical report, International Organization for Standardization, 2021.

[106] ISO Central Secretary. UNI EN ISO 80601 - Medical electrical equipment Particular requirements for basic safety and essential performance of critical care ventilators. Technical report, International Organization for Standardization, 2022.

[107] Hao Jin and Tatsuhiro Tsuchiya. Constrained locating arrays for combinatorial interaction testing. *Journal of Systems and Software*, 170:110771, dec 2020.

[108] T. Kahkonen. Agile methods for large organizations - building communities of practice. In *Agile Development Conference*, pages 2–10, 2004.

[109] Marcin Karcz, Alisa Vitkus, Peter J. Papadakos, David Schwaiberger, and Burkhard Lachmann. State-of-the-art mechanical ventilation. *Journal of Cardiothoracic and Vascular Anesthesia*, 26(3):486–506, jun 2012.

[110] P Kess and H Haapasalo. Knowledge creation through a project review process in software production. *International Journal of Production Economics*, 80(1):49–55, 2002. Innovation of Technology Management.

[111] Dong-Hyun Kim and Hae-Yeoun Lee. Image manipulation detection using convolutional neural network. *International Journal of Applied Engineering Research*, 12(21):11640–11646, 2017.

[112] A. Kornecki and J. Zalewski. Software certification for safety-critical systems: A status report. In *2008 International Multiconference on Computer Science and Information Technology*, pages 665–672, 2008.

[113] D. RIchard Kuhn, James M. Higdon, James F. Lawrence, Raghu N. Kacker, and Yu Lei. Combinatorial methods for event sequence testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, apr 2012.

[114] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *SP 800-142. Practical Combinatorial Testing*. National Institute of Standards and Technology, Gaithersburg, MD, USA, 2010.

[115] Rob J. Kusters, Youri van de Leur, Werner G. M. M. Rutten, and Jos J. M. Trienekens. When agile meets waterfall. In *19th Int. Conf. on Enterprise Information Systems*, 2017.

[116] Neelu Lalband and Kavitha Dwaram. Software engineering for smart healthcare. 8:325–331, 04 2019.

[117] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, 2007.

[118] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *International Journal of Computer Vision*, 127(5):456–476, May 2019.

[119] Nancy Leveson. Are you sure your software will not kill anyone? *Communications of the ACM*, 63(2):25–28, January 2020.

[120] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[121] Yan Li, Chuan-Hoo Tan, and Hock-Hai Teo. Leadership characteristics and developers' motivation in open source software development. *Information and Management*, 49(5):257–267, July 2012.

[122] Jeffrey A Livermore. Factors that significantly impact the implementation of an agile software development methodology. *J. Softw.*, 3(4), 2008.

[123] Ravi Mangal, Aditya V. Nori, and Alessandro Orso. Robustness of neural networks: A probabilistic and practical approach. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '19, pages 93–96, Piscataway, NJ, USA, 2019. IEEE Press.

[124] Roger W Manley. A new mechanical ventilator. *Anaesthesia*, 16(3):317–323, July 1961.

[125] Paul E. Marik and Jim Krikorian. Pressure-controlled ventilation in ARDS: A practical approach. *Chest*, 112(4):1102–1106, October 1997.

[126] John J. Marini and Luciano Gattinoni. Management of COVID-19 Respiratory Distress. *JAMA*, 323(22):2329–2330, JUN 9 2020.

[127] Tony Marks. Accelerating the project. In *The Practitioner Handbook of Project Controls*. 2020.

[128] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5(1), apr 2017.

[129] Martin Mc Hugh, Oisín Cawley, Fergal McCaffcry, Ita Richardson, and Xiaofeng Wang. An agile v-model for medical device software development to overcome the challenges with plan-driven software development lifecycles. In *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*, pages 12–19. IEEE, 2013.

[130] Jim A McCall, Paul K Richards, and Gene F Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[131] Martin McHugh, Fergal McCaffery, and Valentine Casey. Barriers to adopting agile practices when developing medical device software. In *International Conference on Software Process Improvement and Capability Determination*, pages 141–147. Springer, 2012.

[132] Martin McHugh, Fergal McCaffery, and Garret Coady. An agile implementation within a medical device software organisation. In Antanas Mitasiunas, Terry Rout, Rory V. O'Connor, and Alec Dorling, editors, *Software Process Improvement and Capability Determination*, pages 190–201, Cham, 2014. Springer International Publishing.

[133] Bertrand Meyer. *Agile!* Springer International Publishing, 2014.

[134] Microsoft. Pict github repository. available online at: `https://github.com/microsoft/pict`.

[135] Nils Brede Moe, Torgeir Dingsøyr, and Knut Rolland. To schedule or not to schedule? an investigation of meetings as an inter-team coordination mechanism in large-scale agile software development. *International Journal of Information Systems and Project Management*, 2018.

[136] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.

[137] Plinio P. Morita, Peter B. Weinstein, Christopher J. Flewwelling, Carleene A. Bañez, Tabitha A. Chiu, Mario Iannuzzi, Aastha H. Patel, Ashleigh P. Shier, and Joseph A. Cafazzo. The usability of ventilators: a comparative evaluation of use safety and user experience. *Critical Care*, 20(1), aug 2016.

[138] Bacha Munir and Yong Xu. Effects of gravity and surface tension on steady microbubble propagation in asymmetric bifurcating airways. *Physics of Fluids*, 32(7):072105, July 2020.

[139] Srinivas Murthy, Charles D. Gomersall, and Robert A. Fowler. Care for Critically Ill Patients With COVID-19. *JAMA-JOURNAL OF THE AMERICAN MEDICAL ASSOCIATION*, 323(15):1499–1500, APR 21 2020.

[140] Elham Nazari, Mohammad Shahriari, Maryam Edalati, and Hamed Tabesh. Create frameworks from software engineering to health care: A survey article info abstract. *Journal of Biostatistics and Epidemiology*, 01 2019.

[141] NIST. NIST sequence covering array generator. `https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software/event-sequence-testing`/unders, 2016.

[142] Helga Nyrud and Viktoria Stray. Inter-team coordination mechanisms in large-scale agile. In *Proceedings of the XP2017 Scientific Workshops*, 2017.

[143] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, oct 1995.

[144] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, page 298–312, Berlin, Heidelberg, 2011. Springer-Verlag.

[145] Patrizio Pelliccione, Eric Knauss, Rogardt Heldal, S. Magnus Ågren, Piergiuseppe Mallozzi, Anders Alminger, and Daniel Borgentun. Automotive architecture framework: The experience of volvo cars. *Journal of Systems Architecture*, 77:83 – 100, 2017.

[146] Justyna Petke. Constraints: The future of combinatorial interaction testing. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. IEEE, may 2015.

[147] Justyna Petke, Myra B Cohen, Mark Harman, and Shin Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9):901–924, 2015.

[148] Tài Pham, Laurent J. Brochard, and Arthur S. Slutsky. Mechanical ventilation: State of the art. *Mayo Clinic Proceedings*, 92(9):1382–1400, sep 2017.

[149] Candice Quist. Benefits of blending agile and waterfall project planning methodologies. Technical report, University of Oregon, 2015.

[150] Jef Raskin. Comments are more important than code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation. *Queue*, 3(2):64–65, 2005.

[151] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, sep 2017.

[152] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, September 2020.

[153] Elvinia Riccobene and Patrizia Scandurra. A formal framework for service modeling and prototyping. *Formal Aspects Comput.*, 26(6):1077–1113, 2014.

[154] Adrian Rosebrock. Breast cancer classification with keras and deep learning. `https://www.pyimagesearch.com/2019/02/18/breast-cancer-classification-with-keras-and-deep-learning/`, 2019.

[155] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[156] J. M. Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5), 1981.

[157] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[158] R. A. Schrenker. Software engineering for future healthcare and clinical systems. *Computer*, 39(4):26–32, 2006.

[159] Jan Schroeder, Christian Berger, and Thomas Herpel. Challenges from integration testing using interconnected hardware-in-the-loop test rigs at an automotive oem: An industrial experience report. In *Proceedings of the First International Workshop on Automotive Software Architecture*, WASA '15. Association for Computing Machinery, 2015.

[160] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. ACM Press, 2011.

[161] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. Incremental learning of object detectors without catastrophic forgetting. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, oct 2017.

[162] George Spanoudakis and Andrea Zisman. Software Traceability: A Roadmap. In *Handbook Of Software Engineering And Knowledge Engineering*. 2005.

[163] Viktoria Stray, Nils Brede Moe, and Rashina Hoda. Autonomous agile teams: Challenges and future directions for research. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 2018.

[164] Anusuyah Subbarao and MNR Mahrin. A systematic review of coordination approaches and indicators in global software development projects. *Journal of Advanced Research in Dynamical and Control Systems*, 11(10), 2019.

[165] Tianze Sun, Xiwang He, Xueguan Song, Liming Shu, and Zhonghai Li. The digital twin in medicine: A key to the future of healthcare? *Frontiers in Medicine*, 9, July 2022.

[166] Tolga Tasdizen, Mehdi Sajjadi, Mehran Javanmardi, and Nisha Ramesh. Improving the robustness of convolutional networks to appearance variability in biomedical images. In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. IEEE, apr 2018.

[167] Rachel Tzoref-Brill and Shahar Maoz. Modify, enhance, select: Co-evolution of combinatorial models and test plans. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 235–245, New York, NY, USA, 2018. Association for Computing Machinery.

[168] Mark Utting, Bruno Legeard, Fabrice Bouquet, Elizabeta Fourneret, Fabien Peureux, and Alexandre Vernotte. Chapter two - recent advances in model-based testing. In Atif Memon, editor, *Advances in Computers*, volume 101 of *Advances in Computers*, pages 53–120. Elsevier, 2016.

[169] David A. van Dyk and Xiao-Li Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001.

[170] João Varajão. Software development in disruptive times: Creating a software solution with fast decision capability, agile project management, and extreme low-code technology. *Queue*, 19(1), 2021.

[171] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30(5):273–282, sep 2005.

[172] Ferdinand Wagner. *Modeling Software with Finite State Machines*. Auerbach Publications, May 2006.

[173] Michael Wagner, K. Kleine, Dimitris Simos, R. Kuhn, and R. Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index. In *International Workshop on Combinatorial Testing (IWCT 2020)*, 3 2020.

[174] James M. Walter, Thomas C. Corbridge, and Benjamin D. Singer. Invasive mechanical ventilation. *Southern Medical Journal*, 111(12):746–753, dec 2018.

[175] Beilun Wang, Ji Gao, and Yanjun Qi. A theoretical framework for robustness of (deep) classifiers against adversarial samples. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*, 2017.

[176] J. H. Weber-Jahnke, M. Price, and J. Williams. Software engineering in health care: Is it really different? and how to gain impact. In *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*, pages 1–4, 2013.

[177] Cort J. Willmott and Kenji Matsuura. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research*, 30(1):79–82, 2005.

[178] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[179] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Heldal. Improving the consistency and usefulness of architecture descriptions: Guidelines for architects. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 151–160, 2019.

[180] Rebekka Wohlrab, Patrizio Pelliccione, Eric Knauss, and Mats Larsson. Boundary objects and their use in agile systems engineering. *Journal of Software: Evolution and Process*, 31(5), 2019.

[181] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375, 2013.

[182] Linbin Yu, Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Ram D. Sriram, and Kevin Brady. A general conformance testing framework for ieee 11073 phd's communication model. In

Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments, PETRA '13, New York, NY, USA, 2013. Association for Computing Machinery.

[183] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons, 2019.

[184] David R. Ziehr, Jehan Alladina, Camille R. Petri, Jason H. Maley, Ari Moskowitz, Benjamin D. Medoff, Kathryn A. Hibbert, B. Taylor Thompson, and C. Corey Hardin. Respiratory pathophysiology of mechanically ventilated patients with covid-19: A cohort study. *American Journal of Respiratory and Critical Care Medicine*, 201(12):1560–1564, 2020. PMID: 32348678.