Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN
AUTOMATICA I RICERCA OPERATIVA

Ciclo XXVI

Settore Concorsuale di afferenza:	**01/A6**
Settore Scientifico disciplinare:	**ING-INF/04**

Discrete Event Systems based Design Patterns for Diagnosability Analysis of Automated Manufacturing Systems

Presentata da:	**Dmitry Myadzelets**
Coordinatore Dottorato:	**Daniele Vigo**
Relatore:	**Andrae Paoli**

Esame finale anno 2014

Discrete Event Systems based Design Patterns for Diagnosability Analysis of Automated Manufacturing Systems

by

Dmitry Myadzelets

Submitted to the
Department of Electrical, Electronic and Information Engineering
"Guglielmo Marconi" (DEI),
Center for Research on Complex Automated Systems (CASY)
in fulfillment of the requirements for the degree of

PhD

at the

UNIVERSITY OF BOLOGNA

March 2014

Thesis Supervisor: Andrea Paoli

XXII Cycle

2011 – 2014

Discrete Event Systems based Design Patterns for Diagnosability Analysis of Automated Manufacturing Systems

by

Dmitry Myadzelets

Submitted to the Department of Electrical, Electronic and Information Engineering
“Guglielmo Marconi” (DEI),
Center for Research on Complex Automated Systems (CASY)
in fulfillment of the
requirements for the degree of
PhD

Abstract

The main goal of this thesis is to facilitate the process of industrial automated systems development applying formal methods to ensure the reliability of systems. A new formulation of distributed diagnosability problem in terms of Discrete Event Systems theory and automata framework is presented, which is then used to enforce the desired property of the system, rather then just verifying it. This approach tackles the state explosion problem with modeling patterns and new algorithms, aimed for verification of diagnosability property in the context of the distributed diagnosability problem. The concepts are validated with a newly developed software tool.

Thesis Supervisor: Andrea Paoli
Title: Professor

Contents

List of Figures

List of Tables

Introduction

Modern society is characterized by the growing demand for the use of automated systems in the everyday life, and these systems require software engineering with constantly increasing complexity. Industrial automation software engineering has traditionally been a field with the most strict requirements and highest standards applied. However, it is still a common practice for an automation software design to consist of writing a ladder logic for programmable logic controller (PLC) with partially and ambiguously determined specifications, without clearly defined software architecture, and no formal verification of the system. Reflecting the importance of the software and a growing ratio of the software cost to the costs of machinery, engineers and researches put a lot of effort toward facilitating the development and maintenance of software, increasing its performance and reliability while decreasing the cost of its life cycle. One of the major established trends is use of standardised component solutions for industrial automation systems aiming at portability, reusability, interoperability and reconfiguration of applications. Another tendency is the growing number of formal methods - mathematical approaches supporting decisions making process during systems design and operations.

Design Process of Industrial Systems

According to the standard ISO/IEC 12207 [8] the industrial software development process (life cycle process) can be structured into six stages: requirements specification, software design, implementation and integration, testing, deployment and maintenance. Starting from the implementation the stages mainly depend on a particular vendor of hardware and a dedicated software - supervisory control acquisition system (SCADA). There are hundreds of major producers of automation hardware and software, but the variety of ways the developers can write programs is limited by the standards, such as IEC 61131, making them less error prone and more reusable. In other words, this sequence of stages is quite mature.

The stage of requirements specification consists of analyzing, documenting and validating the needs and conditions for technological process, as well as rules, constrains and policies for the plant hardware. The most known standard used at this stage is the Unified Modeling Language (UML), developed by the Object Management Group (OMG) Technology Standards Consortium. This standard facilitates also the next stage, software design, with a Model-Driven Architecture (MDA) approach. The industrial software design stage adopts numerous methods, such as component based approach, object-oriented and aspect-oriented programming, and software product line, etc (see [58] for an extensive survey of the state of the art for software engineering methods in industrial automation).

Problems Statement

The major drawback of the aforementioned industrial software development process is that it is mainly designed for humans, whereas the reliability and other important properties of systems in large part depend on machine-oriented formal verification methods. Formal methods apply mathematically-based techniques to the development of systems, from the specification level to the implementation level. These methods proved to be effective, especially for safety critical systems, but due to their mathematical nature and lack of the supporting tools the use of formal methods in industrial practice is not common yet. The problem is that mathematical notation requires to have additional knowledge on the part of the development engineers, creating a psychological barrier for them, and also that the 'de facto' development process does not incorporate a formal representation.

A way to overcome the above problem is to mix formal techniques with "standard-based" approaches/tools which are already adopted in industry. If a non formal (or semi-formal) systems representation has the ability to be translated into a completely formal form, then mathematical techniques can be applied. Examples of such approaches are [31], where authors extend UML representation with a "collaboration diagram" and translate it to extended hierarchical automata, [51] which formalizes UML into Dirac structures, and [63] where the authors translate Simulink diagrams into input/output extended finite automata.

Even though the translation into a mathematical representation gives the opportunity to apply formal techniques, the requirements for the system's properties should again be expressed in a formal way by engineers, which raises the above mentioned problems. Thus, despite of the fact that some successful reports can be found in literature, progress along these lines seems minimal .

Another solution to overcome the problem is the development of new approaches which incorporate advantages of both methodologies, the one suitable for humans, and the other using formal techniques. Nowadays, when the growing complexity of automated systems imposes requirements which can be met only by formal methods, the relevant response may be the creation of modeling techniques when the models already encapsulate their formal representations. Thus, the underlining mathematical nature would be "hidden" from engineers, and verification of important properties can be performed automatically. An example of such approach is presented in [50] where the authors propose a library of general UML blocks where each block is corresponded to a predefined automaton. The simple blocks can be used then to construct more complex entities while their formal representation can be achieved automatically by composition of the predefined automata.

The necessity of composition of modules formal representations for the sake of verification of some important properties gives rise to another problem, which is inseparable from the formal methods, - the problem of so-called *state explosion.* This problem can be solved by development of mathematical techniques which efficiently exploit the modular nature of the systems such that the correspondent computational burden remains at an acceptable level.

As soon as the quantity and quality of mathematical methods and their implementations is sufficient for the current complexity of automated systems, and these methods can be encapsulated into integrated development tools as a first-class citizen, seamlessly providing the power of formal approaches through the easy-to-understand visual modeling process, the development process of automated systems will move to the next stage of its evolution.

Contribution

This work aims to facilitate the process of industrial automated systems development applying formal methods to ensure the reliability of systems. From many existing problems, for this work has been chosen the problem of verification of system's design in terms of diagnosability. Diagnosability problem answers the question: either the given system is ready for fault diagnosis or not. According to the IEC vocabulary [7] the *fault diagnosis* are "actions taken for fault recognition, fault localization and cause identification", and a *fault* is "the state of an item characterized by inability to perform a required function".

To achieve the goals, this work exploits Discrete Event Systems theory [26] and its automata framework. The theory of automata is chosen due to its relative simplicity. Indeed, it has just two basic entities (state and event) and intuitively intelligible graphical representation. Taking apart a language theory, which is necessary for research, the automata framework does not require much additional mathematical knowledge for automation engineers. This makes it the first candidate as a formal mathematical tool for future general purpose integrated development environments (IDE) in automation industry.

The main contributions are following:

- A new formulation of distributed diagnosability problem. This formulation is then used to enforce the desired property of the system, rather then just verifying it. This approach tackles the state explosion problem which arises when the automata framework is applied for complex discrete event systems.

- Presented new structural patterns, in terms of automata framework. These patterns are aimed to decrease computation burden while applying verification algorithms.

- Created algorithms for verification of diagnosability property in the context of the new distributed diagnosability problem.

- Extended the approach of *GeneralizedDevice*, recently developed in the University of Bologna [33], [50]. This work extends it towards formalization of failures representation in order to apply the developed algorithms. It allows the embedding of a formal approach into a higher level of the systems' representation of "standard" modular design process, similar to one exploiting UML design tools.

- To validate the concepts developed during the research, a software tool was created. This tool exploits web-technologies in order to provide zero-time access to it, the possibility for easy modifications in order to solve other formal problems, and opportunities for further enhancement by community.

Organization

Chapter 1 gives an overview of the problem of fault diagnosis in a real example of an industrial automation plant. It provides a short description of the technological process with a couple of illustrative failures. The types of failures are discussed, and how particular system's failures can be determined, evaluated and treated.

Chapter 2 describes the UML based modeling framework, and how the framework specifies hardware modules with automata. Then the formal framework for discrete event systems is briefly presented, different definitions of diagnosability are explained, and the general idea of new notion of virtual modular diagnosability is given.

Chapter 3 presents the new type of diagnosability - virtual modular diagnosability, in details. It is firstly explained with a simple example of a system consisting of two modules, then a general form of the approach is described. At the end of the chapter algorithms for verification of the virtual modular diagnosability are described.

Chapter 4 introduces a new tool for modeling and simulation with automata. An application of the tool to the example of the real system presented in the Chapter 1 is shown.

The conclusions for the work are given at the end.

Chapter 1

Failure Diagnosis for Industrial Automation Plant

This chapter uses an example of industrial automation plant to present the problem of fault diagnosis. After a short description of the technological process it shows two cases of failures. Then the types of failures are discussed, and what approaches are adopted in industrial and academic practices.

1.1 Case Study of the Automated System

An instance of the real automated systems is exploited to explain the problem of failure diagnosis. It is a dust cleaner at an aluminium smelter. The Figure 1-1 depicts a simplified Piping and Instrumentation (P&I) of the system.

1.1.1 Technological process

For the given system two flows can be distinguished in the technological process. The first is the air flow which is blown trough a series of containers (the figure depicts only one container, for simplicity). Each container is equipped with an electrostatic filter. The second flow is the flow of a dust. The dust comes with the air, sticks to the filter, and then falls down to the bunker. Periodically the dust collected in the

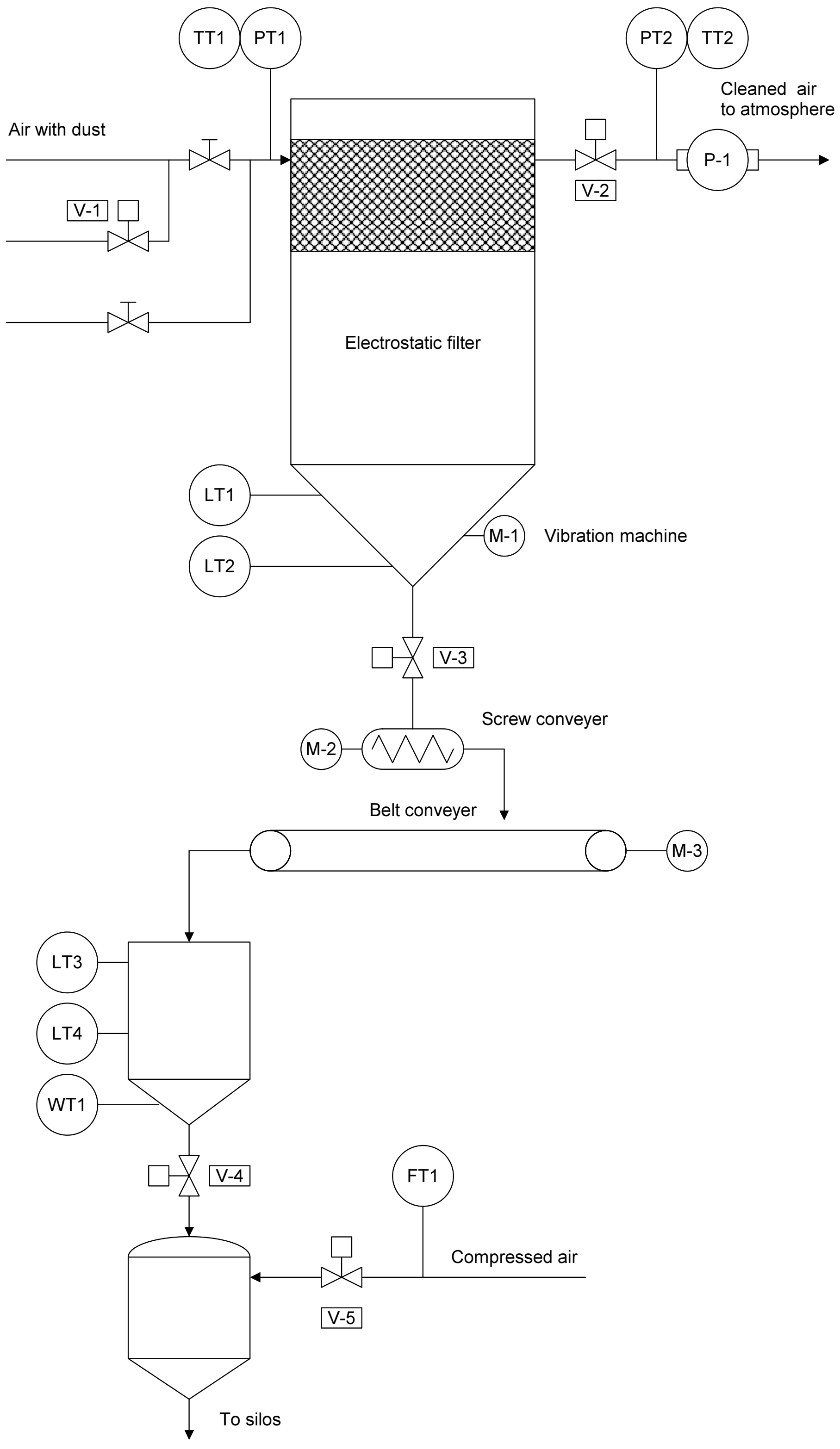


Figure 1-1: Piping and Instrumentation (P&I) diagram of a dust cleaning system at an aluminium smelter

bunker is transported out to a silos.

For the proper dust cleaning, the following parameters of the air flow have to be maintained: a constant difference between the output and the input pressures of the container, and a temperature at the input of the container. The pressure is regulated by the valve V–2, which changes the area flow before the pump P–1. The temperature is regulated by the valve V–1 which changes the area flow of a preheated air. The air at the entrance of the filter is, thus, a mix of the the polluted air and the preheated air. The temperature of this mix should be higher then a certain point such that no condensation inside of the container is guaranteed.

When the amount of the dust collected at the bottom of the container reaches a necessary volume, i.e. it reaches a certain level, the dust is removed by the screw conveyer. The gate valve V–3 is normally opened, since the screw of the conveyer blocks the flow of the dust automatically when it is stopped. The belt conveyer, in its turn, starts to work whenever one of the screw conveyers starts. It delivers the dust to a dust bucket.

The dust bucket collects the dust until a volume of the dust is enough to pass it down, to a pneumatic container. Then the gate valve V–4 opens and the dust fills the container. Then the gate valve V–4 closes and the gate valve V–5 opens allowing the compressed air to come to the container. Under a high pressure the mix of the dust and the air is transported to a silos.

Periodically, in order to prevent sticking of the dust to the walls of the container with electrostatic filters, the vibration machine M–1 has to be switch on for a few seconds.

1.1.2 Control System

The control system consists of a PLC (Siemens Simatic S7–300 [13]), remote acquisition modules and an operator station (with Siemens WinCC software) connected via industrial network. The PLC is mounted near the field equipment, the operator station is located a few hundreds meters away.

The control systems has a few control cycles. Starting to the bottom-up direction, the first cycle controls emptiness of the dust bucket with two level sensors, $LT3$ and $LT4$. When a level of the dust reaches the high level sensor $LT3$, the system starts to empty the bucket with the pneumatic container until a level of the dust reaches the low level sensor $LT4$. The weight meter $WT1$ is used to collect the statistic, and its measurements can be also used to evaluate the amount of the dust in the container. The air flow meter $FT1$ is aimed to collect statistical data too.

The second control cycle regulates the filling of the dust bucket. When the dust level in the bucket is lower then the high level sensor $LT3$, and there is an enabling signal from the upper (with respect to the material flow) control cycle, it turns on the belt conveyer and enables a signal to one of the upper control cycles (each container with its filter has its own control cycle). Either the level of the dust in the bucket reaches the high level sensor $LT3$ or there is no any enabling signals from the upper control cycles, then the belt conveyer stops.

The third control cycle regulates the level of the dust in the container with the filter (the cycle is the same for each container). When the dust level reaches the high level sensor $LT1$, it gives enabling signal to the below control cycle of the belt conveyer and waits for enabling signal from it. After receiving the enabling signal, the control cycle switches on the screw conveyer. Ether the container becomes empty, i.e. the dust level decreases below the low level sensor $LT2$, or the control cycle receives disabling signal from the below control cycle, then it switches the screw conveyer off.

The forth control cycle regulates an order the containers with the filters should be emptied. If more then one container is ready, i.e. the level of the dust reached its high level sensor, then the first container has a priority, since the speed it is filled with dust is faster. Then the second container is emptied and so forth.

The fifth control cycle switches on vibration machines periodically, but only when the container is being emptied.

All the actuators of the system can be in three states: disabled for operation, manual control or automatic control. A signal correspondent to the current state of each actuator is received by the PLC.

1.1.3 Failures of the system

The system can be affected by different types of failures, which may be classified into the following categories with respect to likelihood the failure occurs (the first type has the highers rate of occurrence):

- Failures of the technological process
- Failures of mechanical devices
- Failures of electric and electronic components of the control system

Moreover, the failures can be external or internal with respect to the given system. Yet another type of failure is one when malfunction of the system's behavior is caused by an inconsistent human (operator) intrusion; it has left outside of the scope. Some of the possible failures are described further, according to the above classification.

Failures of the technological process

The technological failures might be caused by deviation of properties of the materials involved in the process. Particularly in this system, a deviation of the dust's or air's humidity can result in that the dust sticks to the filter, to the walls of the container and to pipes, and obstructs the flow of the material. Whereas the temperature of the air (low value of which is the main reason for the condensation) is controlled, the humidity is not measured.

The air, coming to the pneumatic container has neither a temperature nor humidity control. Additionally, the rise of the material humidity may happen due to a leak of rainwater from the building's roof, and etc.

In general, technological failures are hard to predict, since there is usually a little of statistical information for the particular process, if any.

Failures of the mechanical devices

Malfunction of field mechanical devices, probably, is the most frequent reason the automatic systems fail. Firstly, the field devices are exposed to hard environmental conditions, such as temperature, vibration, humidity, electromagnetic field, mechanical impact and etc. Secondly, they are normally complex devices, composed of many other components, and thus the failure rate of a complex device is higher then the highest failure rate of its modules. A common solution adopted in industry is that all the actions performed under each individual device, conditions under with the equipment is used, a time duration it works, and etc. are journalized. These statistical information is later used to plan and perform the maintaining and repairing.

The above presented system has following devices with mechanical parts and parts which can be affected mechanically (e.g. by the flow of a material):

- Butterfly valves V–1, V–2 and gate valves V–3, V–4, V–5. The weakest part of a valve is its actuator, which contains a plenty of moving components. Beside this, the valve's core is subject to abrasion.
- The screw conveyer and the belt conveyer. These devices have a lot of moving components either, where the belt as the weakest part.
- Electric motors M–1 – 3 and pump P–1, which are subject to mechanical and electric damages.
- Weight meter $WT1$, with its fore tension meters which are subject to degradation.
- Fork level meters $LT1$ – 4. The forks are subject to abrasion.
- Pressure meters $PT1$, $PT2$. The drive mechanism of a pressure meter is the subject to stacking, dirt and oil clogging.

All the above potential mechanical failures, the most of the listed above devices contain electronic and electric components, which can have their one failures.

Failures of electric and electronic components

Due to enormous amount electric and especially electronic components used in modern manufacturing industry, it is quite impossible to make a classification of all the failures which might occur, and it is seems useless to have such information at the level of industrial automation, since a fault of a single component most likely results in the failure of entire device it contains. Instead, complex characteristics are used in order to be able to estimate possible faults. One of the well know parameters is the Mean time between failures (MTBF). Another, relatively modern standard IEC 61511 also defines a set of characteristics (MTBF and IEC 61511 are described briefly later).

1.1.4 Examples of the systems' failures

This work does not consider the question of faults prevention. The fact of a fault occurrence has to be assumed, since the problem of the failure is not only that it may lead the necessity to interrupt the technological process, delay for delivery of spare parts and row materials. The major problem is that one failure can sequentially cause other failures which may spread further and so forth, with exponential growth, provoking a catastrophic impact at the end.

As an example, the system presented in this chapter was subjected to some failures observed by the author of this work. These failures are interesting from the point of view of their propagation.

In the first case the high level sensor $LT3$ failed while the system was at the state of refilling the dust bucket. Since one of the conditions the control system has to stops the filling is that the dust bucket is full, it was delivering more and more material. The material overflowed the bucket in great quantity, causing clogging of ventilation holes of the control system (which later resulted in some spare parts change), and blocking entrance to the building. The failure was spotted occasionally; elimination of its consequences took a few days. It was discovered later, that the failure of the fork level sensor was caused by blocking it with a wet material that, in its turn was caused by a high humidity in the building. Thus, the technological failure has propagated to a failure of sensor, and later to a failure of electronic components.

Another example is related not to a failure of particular equipment, but to the control system. In this case the shaft of the motor M–2 was left not connected to the screw conveyer after a maintenance service. Thus, when the motor is switched on, the screw conveyer does not rotates and blocks the flow of material. When the system started to empty the first container, i.e. the control cycle responsible for unload of the container with electrostatic filter had enabling signal to the below control cycle, and the control cycle responsible for the filling of the dust bucket, in its turn, had an enabling signal for the upper control cycle, the system was found infinitely trying to fill the dust bucket from the first container. If not spotted, this failure could lead to the overfilling of the container, clogging of the filters and air channels, and pollution of the environment.

These illustrative examples of failures are showing importance of the failures detection. If a particular single fault can not be detected, at least one the propagating effects should be observed. The earlier a failure is detected the less damaging impact it leads to. The evidence of this fact can be stressed by a numerous examples and case studies: [38], [5], [20] to mention a few.

1.2 Industrial and Academical Analytical Approaches

1.2.1 Prevent, Detect, Isolate and Recover

From the point of view of an automation engineer a failure is an event which has to be, firstly, *prevented*, then *detected* and *diagnosed*. It is clear that nobody can guarantee that a failure can not occur under all circumstances. As it was shown by examples in the previous section, failures also tend to propagate, causing other failures with a greater impact. Thus, a general assumption is that the failures are not avoidable, but the failure propagation may be prevented.

Determinization of all the potential failures is a separate difficult task. This problem is addressed with the different qualitative and quantitative analytical methods at the design stage while the development of a new system.

The next step, after all the possible system's failures are found and evaluated, is to decide if each failure can be detected. The problem is that the effects of a failure may be difficult to observe. There can be a lack of the technological process parameters deviation, lack of equipment (sensors) or time (performance). Even if observed, the failure may be not easy to distinguish from a normal system's behaviour. Moreover, it might happen that the observed effects caused by one failure can not be distinguished (or, in other words, isolated) from effects caused by another failure. These two failures may required different reactions, thus they has to be distinguished. The problem of failure detection and isolation is know as a diagnosability problem.

After a failure is detected and analysed, the system usually is desired to continue operating properly, possibly at a reduced level of performance, but not to fail completely. Such property of systems is called *fault tolerance*. In industrial applications the property of fault tolerance is usually achieved by redundancy, i.e. is by duplication of critical components or functions of a system. The major form of redundancy in manufacturing are:

- Hardware redundancy, such as double module redundancy or triple module redundancy
- Information redundancy, such as error detection and correction methods.
- Functional redundancy, when the same input produces an equivalent output through different media or different principles

For example, in the dust cleaning systems, presented in this work, the three containers with electrostatic filters may be considered as a form of hardware redundancy. A level of dust in the dust bucket can be measured by mean of its weight and by the fork level sensors, thus, providing a form of information redundancy. When the system works in the non-automatic mode, based on decisions of its operator, actuation of motors can be performed both via the software control system and locally by hardware buttons - functional redundancy.

The problem of fault-tolerance is out the scope of this work. We proceed with a brief description of some qualitative and quantitative analytical approaches adopted in industry, such as Failure Mode and Effects Analysis (FMEA), Mean time between failures (MTBF), reliability online databases and IEC 61511 standard. Then a Simulation approach is described as a way for failure validation, and, finally, the formal methods approach. Some of these approaches can be considered as pure industrial ones, while other widely used in both industrial and academic fields. The formal methods is seen as a pure academic response to the problem of failures diagnosis.

1.2.2 Failure Mode and Effects Analysis

FMEA is an exhaustive analysis of systems, an analytical tool to identify, quantify, prioritize and evaluate the risk of possible failures in order to reduce risk of failures, ensure that failures are detectable and prevent failure from happening. Assuming that any failure which was determined can occur, it is necessary to figure out what negative impact of each failure (of type of failures) can be, i.e. what failures must to be detected, what are wished to be detected, and what are not, since the failure detection may be an expensive process in terms of time, increasing complexity and costs of the whole system.

FMEA is described in the IEC 60812 standard, and consists of:

- System analysis - how interactions among systems may fail
- Design analysis - how product design may fail
- Process analysis - how processes that make the product might fail
- Design analysis - focuses on how machinery that perform processes might fail

A chart with steps which are necessary to perform according to FMEA methodology is depicted in Figure 1-2.

FEMA involves reviewing as many components, assemblies, and subsystems as possible to identify failure modes, and their causes and effects. It also involves a lot of human resources: experts and practitioners in particular industrial fields [56]. That is why this tool is considered as a relatively expensive approach, and is adapted mostly for life-critical systems, related to military area, automotive, gas and oil industry, health-care and life-support activity. There are other, less costly but yet similar knowledge based analytical approaches, presented in literature, for instance the adapted for industrial use Process Mapping [27], and State analyses [47].

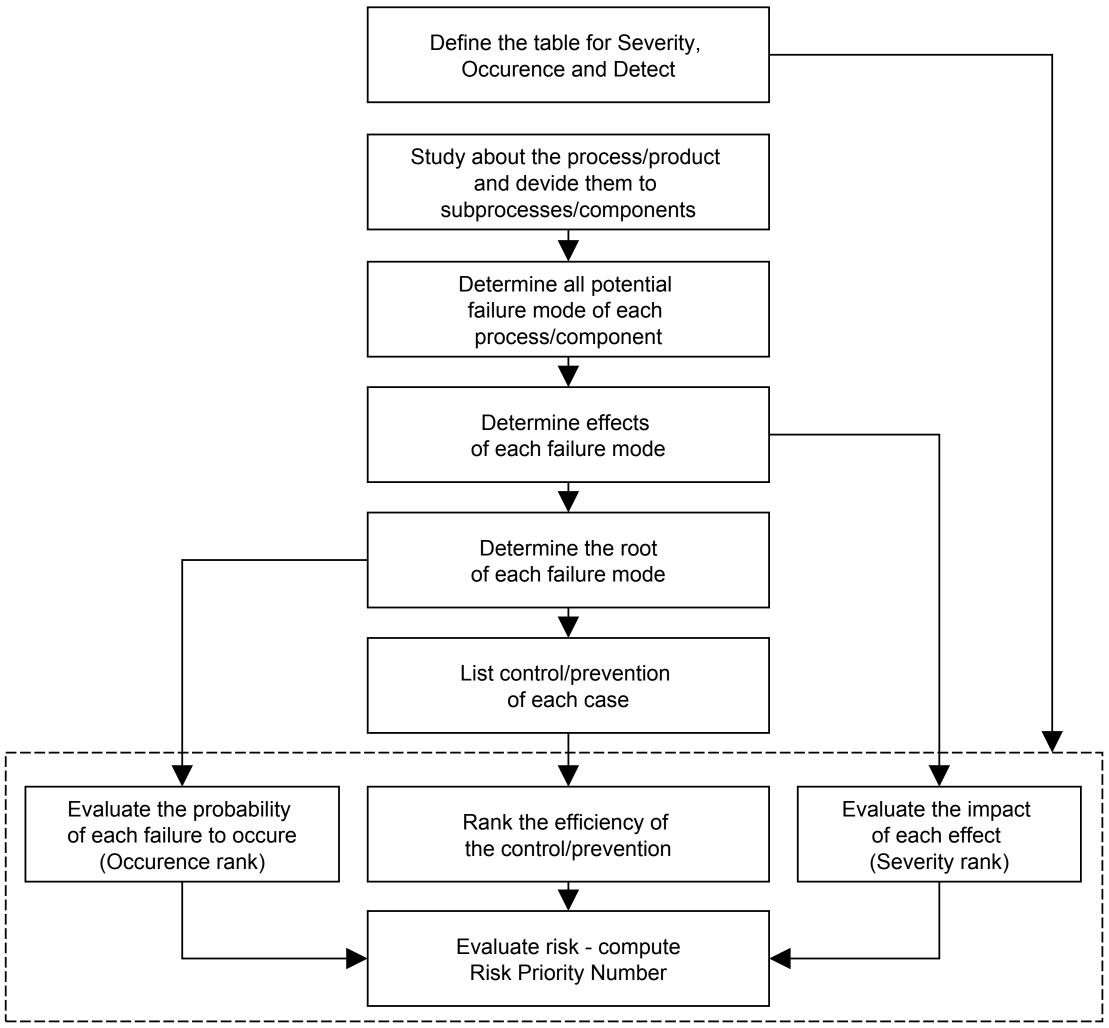

Define the table for Severity, Occurence and Detect
Study about the process/product and devide them to subprocesses/components
Determine all potential failure mode of each process/component
Determine effects of each failure mode
Determine the root of each failure mode
List control/prevention of each case
Evaluate the probability of each failure to occure (Occurence rank)
Rank the efficiency of the control/prevention
Evaluate the impact of each effect (Severity rank)
Evaluate risk - compute Risk Priority Number

Figure 1-2: The necessary steps for FMEA methodology

1.2.3 Mean Time Between Failures

Mean time between failures (MTBF) is a statistical mean value for error-free operation (between two failures) of an electronic device during the normal working life. The MTBF does not apply to an individual component, it always refers to the phase with constant failure rate (i.e. without early or wear failures). The higher the MTBF, the less often the component fails and the more reliable it is. In addition to the MTBF value, the environment and operating conditions should be taken into consideration. If a device is operated under conditions beyond its specification (e.g. at extremely

Table 1.1: MTBF values for some SIEMENS Simatic components

Part Number	Type Description	MTBF (years)
6ES7321-1FF01-0AA0	8 Pt. AC Input	5,4
6ES7314-1AC00-0AB0	CPU 314	22,2
6ES7321-1BH00-0AA0	SM 321, DI 16 x DC24V	28,0
6ES7331-7NF00-0AB0	AI 8 channels	76,0
6ES7951-0FD00-0AA0	Memory Card	90,2
6ES7322-1BH01-0AA0	DO 16 x DC 24V, 0,5 A	105,7
6ES7340-1AH01-0AE0	CP 340	278,3
6ES7960-1AA04-5AA0	Patch cable 1m	394,9
6ES7964-2AA04-0AB0	DP-interface module	703,4
6ES7972-0BA12-0XA0	profibus connector	19120,4
6ES7368-3BB00-0AA0	Cable	317097,9

high ambient temperatures or subject to a massive EMC load), then the MTBF values are no longer valid and large numbers of failures might occur.

This term is based on the assumption that when a failure occurs, the system does not generally remain in the down state, but is renewed or repaired. The time required for renewal or repair (i.e. from the start of the down state to the restoration of the up state) is known as the Mean Time to Repair (MTTR), often also called Mean Down Time (MDT). These characteristics are the most common parameters related to reliability which are provide by manufactures in the documentation for their equipment.

As an instance, the Table 1.1 shows some values MTBF for the equipment used in the previously described system. This data are provided by the manufacture [12].

1.2.4 Knowledge-based Databases

It is common that hardware vendors provide some parameters as MTBF for their equipment, but there is no information about complex types of failures. Since each relatively big manufacture usually has it is own history records for the equipment, an idea to unite such historical data seems to be a relevant responses to the problem. There exists international initiatives which gather the data collected by users of equipment in a centralized database, in order to provide realistic data about different types of technological devices. For example, plant and equipment taxonomies developed by the Process Reliability Database Project (PERD) [2].

1.2.5 Safety standards IEC 61508/IEC 61511

These standards is the quantitative approach for the failures analysis. They define a concept of functional safety as a safety instrumented function calculated for electrical, electronic and programmable electronic systems. The standards require a quantitative proof for the risk, based on calculating the probabilities of dangerous failures. This calculation is carried out for the complete safety loop, consisting of sensor, PLC and actuator. The standards define the following steps for evaluation of safety:

- Risk definition and assessment according to detailed probabilities of failure from sensor over controller to actuator for the overall component life time
- Specification and implementation of measures for risk reduction
- Use of suitable instrumentation (evaluated or certified)
- Periodic test for correct operation of the safety functions

Some of the calculated parameters are following:

- Percentage of failures without the potential to put the safety-related system into a dangerous or fail-to-function state
- Average probability of failure on demand
- Failure rate for all safe detected failures
- Failure rate for all safe undetected failures
- Failure rate for all dangerous detected failures
- Failure rate for all dangerous undetected failures

The calculations are made for every single component of a system. Clearly, a complete quantitative evaluation can be determined only for relatively simple and small devices, such as film resistors, transistors, relays and etc. Characteristics of complex components such as processors and memory for PLC are determined partially.

The complexity of the underlining approach for these standards and a correspondent high cost restricts its application only for so-called Safety Instrumented Systems (SIS) that are designed and used to prevent or mitigate hazardous events, to protect people or the environment or prevent damage to process equipment.

1.2.6 Simulation

Simulation is the imitation of the operation of a real-world process or system over time[21]. In industrial practice it is used as the imitation of the real-life technological processes and equipment. It usually includes generation of a software model which reflects the supposed system's behaviour. It is also a common case when simulations involve hardware devices and entire subsystems, such as networks, sensors and actuators, and even real physical objects and media. Either existing and conceptual systems can be modeled with simulation. The main goal of simulations is validation of system's behaviour while a real system is not accessible, it is being designed, or it may be dangerous to use its processes and machines. Examples of simulation approach can be found in performance optimisation [32], scientific modeling [52], analysis of possible failures [60] and of failure avoidance [59].

The main issue of this approach is how an imitated system is close to the correspondent model, i.e. how the quality and quantity of information provided for the simulation reflects the processes and machines' behaviour in the real life.

1.2.7 Formal methods

The Failure Mode and Effect Analysis is the first step of a system reliability study. The Simulation can augment this analysis with a reality-like experience of failures and of how the automated system would behave. However, only formal methods can ensure reliability of a manufacturing system design [36]. Formal methods apply logic and mathematics to development process of systems and their analysis during operations. The are used for specification, development and verification of software and hardware.

The formal methods have been used in industrial automaton for a half of century, providing distinct field of research, development and application, mainly for control engineering due to its importance. Nowadays, formal methods have their own design programming languages as [53] for instance, and tools as [4], [1] and [10] for example.

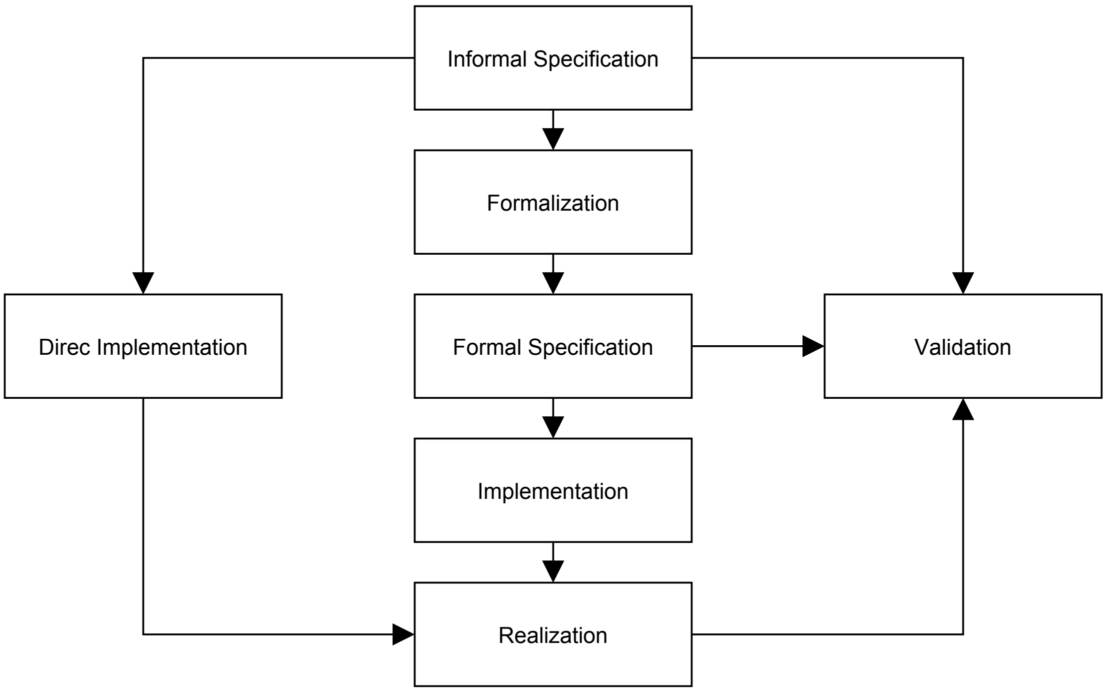


Figure 1-3: Design process of automated systems

The Figure 1-3 depicts a generic model of a manufacturing system design process. The design process of a particular system starts, in most of the cases, with informal description of technological processes. It specifies the system's behaviour in not a strictly semantically and syntactically defined form. Additionally to the verbal form, the description includes P&I diagrams, equations and algorithms expressed in block diagrams. The main problem with informal specifications is that they do not facilitate tests for completeness, unambiguity and consistency of the system.

Until nowadays, the common approach is when after the stage of informal specification immediately begins its direct implementation into a code for PLC with a programming language. Instead, when applying formal methods, the implementation stage follows after formalization of informal descriptions into formal specification. Formalisation is performed by humans, and involves conversion of textual and graphical information into the form of a formal language or another representation (for instance, automata).

Formal specifications can be translated directly into programming languages [41] by a design tool. This step eliminates the error prone interpretation by humans ensuring that the system will behave exactly as it was specified.

Formal specification of the system brings opportunity to apply *verification* techniques to the design process. Verification is used to prove some properties of the system's specification. These properties are independent from the modeled system, such as, for example, existence of deadlocks in the discrete-event systems approach. Verification is closely related to *Model checking* "technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning"[28]. Model checking verification considers a system as state-based model specified using, for instance, automata framework, and specifications of properties written in temporal logic.

A formal verification approach can be a model based or non model based. In model based approaches a model of the studied object is included into analysis. This can be a model of technological process or of manufacturing equipment. In non model based approaches the formal representation of specifications does not include a model of the object. These approaches assume that, literally, everything can happen. An example of this approach is development of a system with a few formally described controllers, where the whole system, i.e. the composition of the concurrent acting controllers needs to be verified for deadlocks absence.

Due to importance of the problem of failures analysis for modern complex automated systems, formal method tools are widely used to support it. Whenever the specification of a system is accessible in a formal form, many formal approaches can be applied for offline failure analyses of the system's design and its online failures monitoring during its operation.

The next Chapter 2 describes an UML model based modeling approach for manufacturing systems design, and how this approach is used to reflect failures, such that it is suitable for formal analysis due to exploiting the automata framework at the internal layer. Then the mathematical notation for discrete event systems is briefly presented, different definitions of diagnosability are given, as well as an idea underlining a new notion of diagnosability which is described in details in the following Chapter 3.

Chapter 2

Diagnosis-oriented Modeling and Verification Framework

This first part of the chapter presents a modeling approach for design of manufacturing systems which exploits idea of decomposition of a system into modules, and presents them in a form of UML blocks. It is shown then, how the modules are represented internally by automata, and how the failures can be modeled. The second part of this chapter gives a necessary introduction into a formal notation for discrete event systems in terms of languages and automata, and describes different types of diagnosability.

2.1 Modeling Framework

2.1.1 Introduction

The fundamental principle "divide and concur" underlies many techniques and approaches either used solely at the level of individual engineers and programmers or developed at the level of international standard committees and organisations. For instances, one can recall object-oriented concepts in traditional programming principles [34], and standards of International Electrotechnical Commission (IEC), particularly the legacy IEC 61131-3 [43] which defines programming languages for programmable logic controllers (PLC), and the modern IEC 61499 [57] which represents a component solution for distributed industrial automation systems aiming at portability, reusability, interoperability and reconfiguration of distributed applications.

Many Integrated Development Environments (IDE) used in industrial world satisfy all the requirements and standards in order to facilitate the process of development, enrollment and maintenance of a complex system. They exploit object-oriented concepts as classes with their instances, inheritance and encapsulation. However, these concepts were not initially created bearing in mind the purpose of formal verification. As a consequence, the tools widely used in industry nowadays do not generally support approaches for DES raising in the scientific world. As a response for the growing demand for intersection of classical development process with formal techniques many researches attempt to create instrumental and theoretical bridges. The most promising results seem to be reached in formalizing Unified Modeling Language (UML), Statecharts, and general purpose programming languages as C.

In [42] the authors introduce a framework which allows to convert an UML model into the formal languages described in the form of specification. The framework can be applied only for a subset of UML diagrams. The resulting formal specification can be used for model checking and simulation, using existing tools aimed for this purpose. Figure 2-1 depicts the instance of an UML diagram and its representation by a Mealy automaton which, in its turn, can be transformed into a classical automaton. In [24] the authors develop a framework to transform UML diagrams and Statecharts into Computation Tree Logic (CTL) which is the used in a model checking tool. Other works to mention are [31], where UML diagrams are mapped to automata, [35] where the authors concentrate on converting PLC languages to UML and then to automata. Even a such known widely spread tool as Matlab [9], used by both practitioners and theoreticians, also has to be tweaked in order to get its modeling ability closer to a formal representation. Examples of such attempts are [46] and [40]. An instance of model checking tool, which is exploited by many of the approaches mentioned above, the most notable one is NuSMV [10] which automatically verifies properties of finite-state systems, expressed in CTL, Linear Temporal Logic (LTL) or Property Specification Language (PSL).

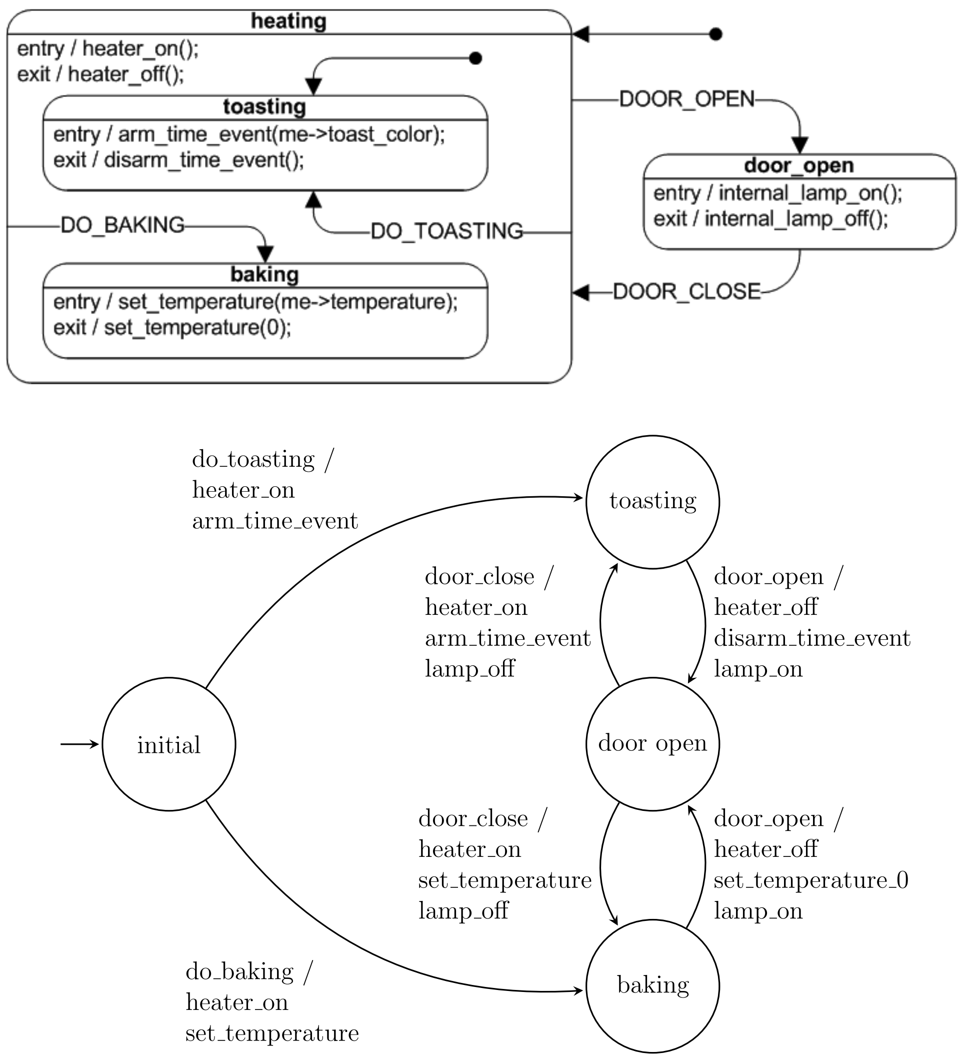

heating
entry / heater_on();
exit / heater_off();
toasting
entry / arm_time_event(me->toast_color);
exit / disarm_time_event();
DO_BAKING
DO_TOASTING
baking
entry / set_temperature(me->temperature);
exit / set_temperature(0);
DOOR_OPEN
door_open
entry / internal_lamp_on();
exit / internal_lamp_off();
DOOR_CLOSE
do_toasting / heater_on arm_time_event
toasting
door_close / heater_on arm_time_event lamp_off
door_open / heater_off disarm_time_event lamp_on
initial
door open
door_close / heater_on set_temperature lamp_off
door_open / heater_off set_temperature_0 lamp_on
do_baking / heater_on set_temperature
baking

Figure 2-1: Example of an UML Statechart diagram (at top) and its representation by the Mealy automaton

Quick analysis of the aforementioned attempts to bridge the most widely used object-oriented IDEs and their underlining modeling technologies shows that they solve the problem of using formal methods by system's developers only partially. The reason is, as it was said before, that those concepts and tools had been developed before formal methods became useful in practice, and they simply can not be merged with modern formal approaches. The problem lays, firstly, in the ambiguity of legacy methods, their freedom of interpretation. It appears when, for instance, one tries to convert an UML diagram to automata. Second, the problem comes from the fact, that producers of development tools have created a wide range of derivatives of one standard, which, at one hand, satisfied requirements of the market and, on the other hand, made these implementations incompatible to each other. Now, when formal approaches both, have reached enough theoretical level, and have growing demand due to the high complexity of currently appearing systems, there is a need for new approaches and tools that exploit formal methods as an underlying concept.

One of the approaches which uses the object-oriented principles, and focuses on formal verification is presented in [50]. The approach introduces a modeling entity called *Generalized Device* and shows how it can be used for formal verification in the DES framework. The approach is a refined architecture of a concept of *Generalized Actuator* for industrial automated systems [33]. The next section gives a brief introduction into both the concept and the refined approach.

2.1.2 Generalized Actuator and Generalized Device

Generalized Actuator

The GA approach aims to reduce complexity of the modeling process using standardized object-oriented concepts of engineering. An idea of GA is to identificate essential patterns in automated manufacturing systems, and design control software entities suitable for formal methods of DES theory. The GA approach gives a modeling framework and defines a design procedure with following characteristics:

- Encapsulation of "actuation mechanism";
- Hierarchical representation of a plant;
- Separation of control policies from actuation mechanisms in each GA;
- Support of visualization of plant's hardware;
- Interoperability and reusability of GAs.

The GA implements a concept of abstraction. It implies that all the actions of a complex process, from the high-level point of view, can be decomposed into sets of two kinds: *Do-Done* actions and *Start-Stop* actions.

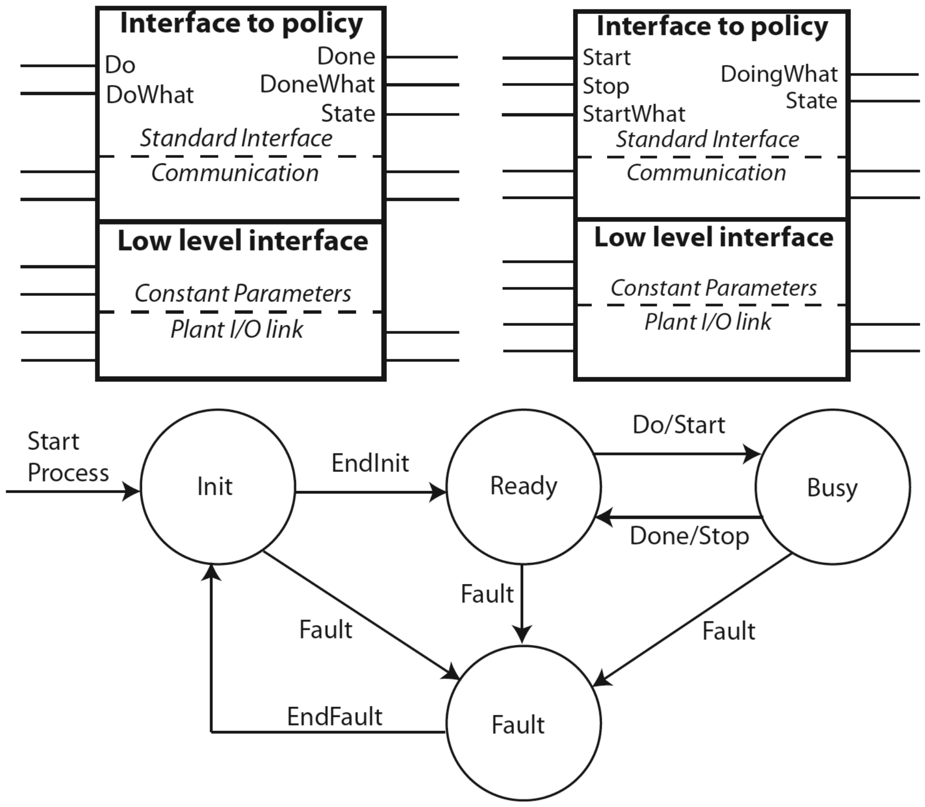


Figure 2-2: GA with two types of interfaces (top) and its underlining formal representation by the automaton (bottom)

The Do-Done GA reflects a process which starts by an external (with respect to this process) “Do” command, and continues until an internal (with respect to this process) decision to terminate is taken. The duration can be finite or infinite. After its termination the process issues “Done” event. The instance of a process which can be abstracted by this type of GA is the opening of a valve: the command “Do” is issued to open the valve; when it opens, the event “Done” occurs.

The Start-Stop GA is an abstraction of a process which starts by an external “Start” event, and continues until an external “Stop” event occurs . Again, the duration can be finite or infinite. The instance of a such process is a temperature regulation with a heater: the “Start” event switches on the heater; when temperature reaches a required level the “Stop” event switches it off.

The above described two types of GA can be implemented as Functional Blocks (FB) in terms of IEC 61131-3 standard, as depicted in Figure 2-2 [1]. Besides the essential inputs and outputs necessary to perform the above described types of actions, the blocks have other inputs and outputs related to the low level of processes, and to some additional information related to the high level.

Internally, each type of GA is represented by the automaton, as depicted on the figure. Each state of the automaton can abstract a set of states of other automata, i.e. GAs contain a hierarchical structure of automata.

A design procedure, according to the framework of GA can be described by the following steps:

- Identification of distinguished processes and their basic actions;
- Classification of actions into Do-Done and Start-Stop actions;
- Definition of GAs for two kinds of actions;
- Assigning low level information and additional information to each GA;
- Design of internal hierarchical structure and logic of automata for each GA.

The last two steps in the above described design procedure encapsulate a logic specific for a given process. Potentially, the complexity of internal implementations of GAs is not limited. Thus, the functional blocks correspondent to the GAs can became very specific, such that it would be possible to reuse them only at the same plant, or for a technologically similar processes. A designer of a GA must be aware of this problem, and maintain simplicity of the internal implementation. Instead of increasing the complexity of internal structure of a GA, one can create other GAs. However, this process relies solely on decision of a particular designer for a system.

The design problem, described above, can be reformulated as that a GA abstracts a process at its high level, but does not abstract it at its low level. The architectural approach of Generalized Device (GD) is aimed to solve this problem.

[1] The image is borrowed from [33]

Generalized Device

The idea of a Generalized Device is based of the observation that the most of different low level field devices can be viewed as only two kind of abstract devices: either single acting device or double acting device. Each kind of devices has an actuator and also may have sensors.

A single acting device has only one command (input) to perform a “forward” or “on” action. The “backward” or “off” action is performed by the device itself, implicitly ore explicitly. Examples of single acting devices are: a cylinder with a return spring, a contactor itself or together with any controlled equipment (electric motor, heater, lighting, etc).

A double action device has one command to activate the device and another command to deactivate it. Simultaneous commands on inputs of the device are usually prohibited or have now effect on a state of the device. The instances of double action devices are: a double acting cylinder, an auxiliary relay or any start-stop device which may itself be a composition of two single acting devices.

The common feature of the both, single and double acting devices is that they have two clearly distinguished states. In order to detect what the current state of a device stays in, it is usually equipped with one or two sensors. In case of two sensors each of them gives the feedback for the correspondent state. In case of one sensor one level of its signal corresponds to the first state of the device, another level - to the second one. A state of the device in the case when it has no sensors is implied according to a previous command. Concluding, a classification of the devices in this approach is the following:

- Single acting device
  - with no feedback
  - with single feedback
  - with double feedback
- Double acting device

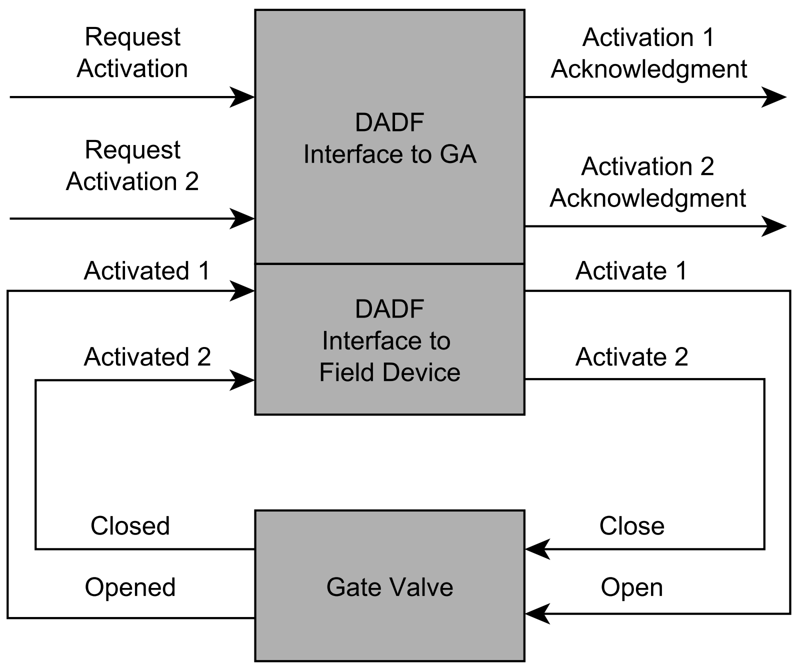


Figure 2-3: Device with Double Actuation and Double Feedback (DADF) and its connection to a physical device (valve)

- with no feedback
- with single feedback
- with double feedback

As an instance, the Figure 2-3 depicts the device with double activation and double feedback. It abstracts a physical device, the valve in this example, for a higher level GA.

The approach of generalized actuators and devices allows to construct complex automated systems using simple predefined models, as it is shown at the Figure 2-4.

Due to representation of the system's models by automata, various DES techniques can be applied in order to define and solve control and diagnosis problems for entire system. To preserve the level of abstraction, however, additional information necessary for a required problem has to be encapsulated into modules. Particularly, possible failure behaviour should be incorporated into modules for the diagnosability problem.

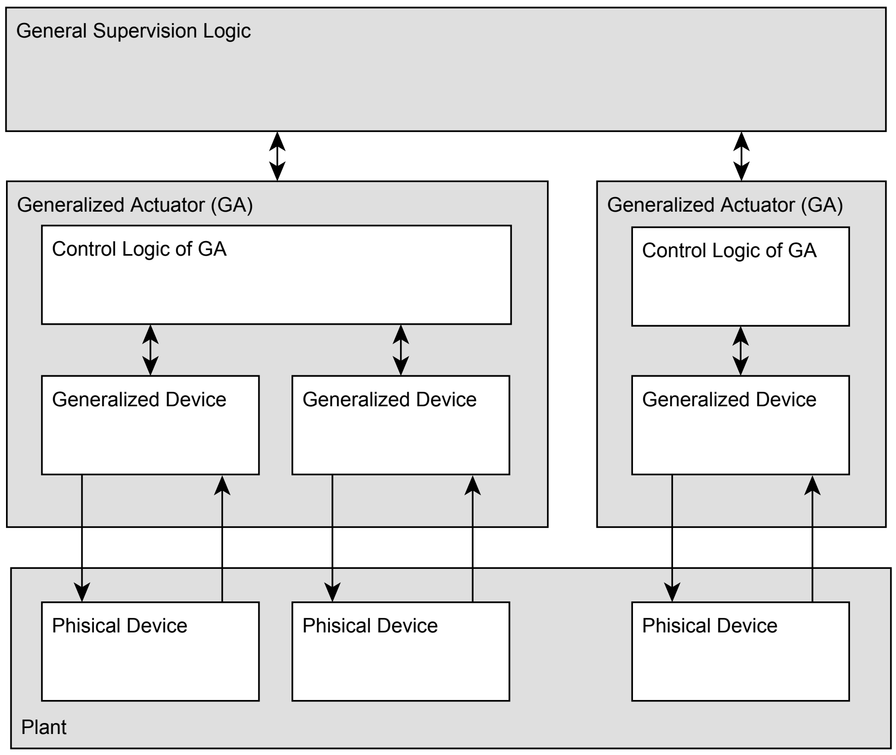


Figure 2-4: Hierarchical relation of system's components: high level logic of plant, GAs, GDs and physical devices

The next section proceeds with a description of how the approach of generalized device is used for modeling and analyses of faults.

Generalized Device and Failures Diagnosis

This section briefly describes an approach for building structured formal models, presented in [49]. The methodology extends the approach of generalized device to facilitate diagnosis verification and monitoring problems. The authors present each GD as a composition of automata, where automata reflect control logic and a model of physical device, decomposed into sensors, actuators, components and their constrains. For the sake of simplicity, the approach is presented further with a small illustrative example. The example shows the idea of decomposition with two components of a hypothetical device (think of a valve from the system given in Chapter 1), their constrains and how failures are modeled.

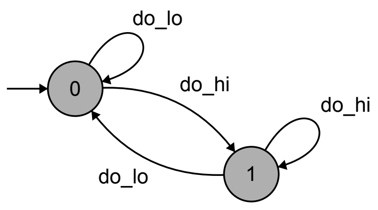


Figure 2-5: Automaton representing digital output

Table 2.1: Correspondence of automata event labels to digital output signals

Label	Digital output signal state
do_lo	Digital output signal is low
do_hi	Digital output signal is high

The Figure 2-5 depicts a model of digital output (a model of digital input would be equal). The automaton has two states, reflecting the low and high levels of signal. The event labels of the automaton are correspondent to digital signal states as shown in the table 2.1. The states of a signal are treated as events because they are seen as readings (two values) of a variable (either it is output or input) in a program in PLC, while the cyclical program execution.

In major of cases, a digital output is linked to an actuator, there is an intermediate signal amplifier, e.g. an electromagnetic relay. It acts also as an galvanic isolation of the PLC and the field device.

In this example, the behaviour of the relay is related to the behaviour of digital output, i.e. it is constrained by the digital output. A model of the relay, a model of constrain and the resulting composition of all the aforementioned components is depicted in the Figure 2-6.

s

For illustration of a failure lets pick one of the possible relays failures, e.g. broken coil. The correspondent model of the relay with this failure is depicted in the Figure 2-7 (event $f0$ states for the failure event). Similar to the this failure, a "stuck on" relay's failure can be modeled.

As the the approach of GD assumes, almost any field device can be decomposed into simple components, as ones described in the above example. The failure models

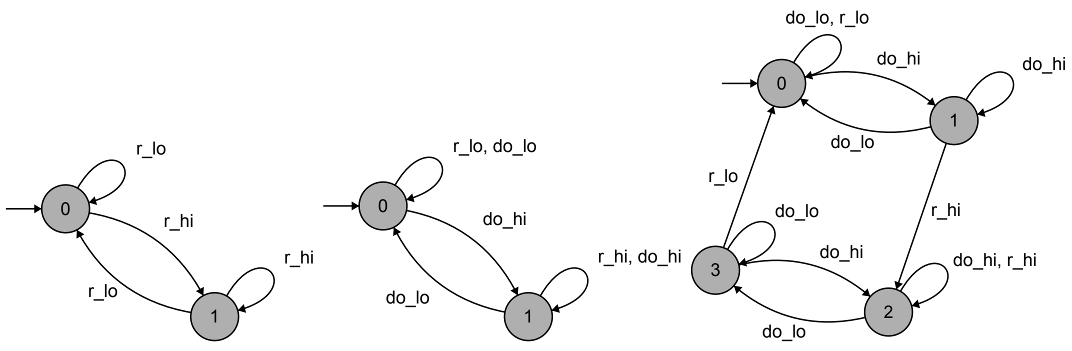


Figure 2-6: Automata models of Relay (top left, constrains (top right), and result of composition (bottom)

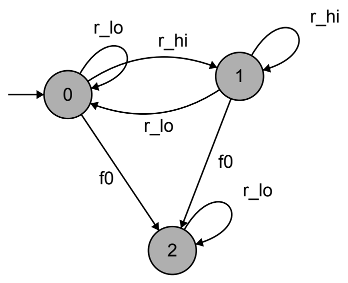


Figure 2-7: Failure model of the relay (example of the broken coil)

for such components can be easily determined. More complex components can be then composed by using the simple models and constrain automata. Thus, a set of predefined models can be exploited for building a library of GD. Since all the models include failure behaviour, formal diagnosability analysis techniques can be applied.

In the Chapter 3 a part of the work is devoted to an extension of the above failure modeling approach such that a failure behaviour can be modeled without failure events, by state marking. It will be shown how a failure behaviour can be separated from a non-failure one automatically, for further diagnosability analysis.

2.2 Discrete Event Systems Framework for Diagnosability Verification

2.2.1 Introduction

Discrete Event Systems (DES) are systems, the dynamic of which is characterized by asynchronous occurrence of *events*. An event is a fundamental concept which can be viewed and described as "something happened", either in systems designed by humans or in nature. Events have no property of continuation, they are instant, and can be observed only at discrete points in time. The second fundamental concept, characterizing a DES, is a state. A state is viewed as a result of *temporally ordered* discrete events, occurred starting from a moment when the system was at its *initial state*. The initial state of a DES characterizes the system before occurrence of any event. In reality, this characteristic may be seen as an assumption or "agreement" for the given system. Being at a certain state, when an event occurs, the system makes a *transition* to another state.

The examples of events are: switching a light on, pressing a button, a moment of rising a temperature above a certain level. The correspondent states are: the light is on, the button is down, the temperature is high.

Historically, all the system's events are thought as of an *alphabet*. Temporally ordered, they form *words* which, consequently, form a *language*. At this level of abstraction DES are studied by the *language theory* [19].

Formal language theory is closely related to *Automata theory* [37]. Automata are used as one of the modeling formalisms for DES. A single automaton can be represented as a directed graph, and it is, probably, the simplest way a complex system's behaviour can be graphically and formally described and conceived by humans.

The next section gives the preliminaries from theories of languages and automata, used in this document.

2.2.2 Notation

The notation used in this document is the one in [26]. For the benefit of the reader we place here the most essential notation necessary for understanding of the further material.

Let Σ be a finite set of events. A sequence of events is a string. Σ^* denotes a set of all finite strings over Σ. $L \subseteq \Sigma^*$ is a language over Σ. Given strings s and t, st is their concatenation. Given strings s and w, w is a prefix of s if exists t such that $wt = s$. Prefix closure of L, denoted by $\overline{L}$ is a set of all prefixes of all the strings in L. If $\overline{L} = L$ then L is prefix-closed. The post language of L after a string s is denoted as L/s, i.e. $L/s := \{t \mid st \in L\}$. We write $\sigma \in s$ if the event $\sigma \in \Sigma$ appears in the string $s \in \Sigma^*$. If $\{s\}$ is a singleton, we write s for operations on languages.

An automaton G is a tuple

$$G := \langle X, \Sigma, \delta, x_0, X_m \rangle ,$$

where X is a set of states, $x_0 \in X$ is an initial state, $X_m \subseteq X$ is the set of marked states, and $\delta : X \times \Sigma \to X$ is the transition function. We say a language $L := \mathcal{L}(G)$ is generated or recognized by the automaton G. In this paper we assume that for each language there is always a correspondent automaton, and vice versa. The marked language $L_m \subseteq L$ is intended to make a part of the automaton's behaviour distinguishable in a certain context.

Some events of DES can not be observed. To reflect that the set of events Σ is partitioned into disjointed sets of observable events Σ_o and not observable events Σ_{ou}, i.e. $\Sigma = \Sigma_o \dot{\cup} \Sigma_{ou}$. $M : \Sigma^* \to \Sigma_o^*$ denotes natural projection that erases unobservable events. The correspondent inverse projection is $M^{-1} : \Sigma_o^* \to 2^{\Sigma^*}$.

Let $I := \{1, 2, \ldots, n\} \subset \mathbb{N}$ be an index set. A system is defined by a set of automata $\{G_{i \in I}\}$ and a correspondent set of languages $\{L_{i \in I}\}$. We use the term *local* in context of the automata and languages from these sets. The *global* language of the system is defined by the parallel composition [26] of its local languages: $L :=\|_{i \in I} L_i$. The natural projection for the local languages is defined as $P_i := \Sigma^* \to \Sigma_i^*$. We extend it to the system's languages as follows: $P_i(L) := \{s \mid s \in L_i\}$, and $P_i^{-1}(L_i) := \{s \mid s \in L, P_i(s) \in L_i\}$, $i \in I$. We also define natural projection over observable events for the index set: $M_i : \Sigma_i^* \to \Sigma_{i,o}^*$, natural projection over common events: $P_{i,c} : \Sigma_i^* \to (\Sigma_i \cap \bigcup_{j \neq i}^{I} \Sigma_j)^*$, and natural projection over observable and common events: $P_{i,co} : (\Sigma_i \cup \Sigma_o)^* \to (\Sigma_o \cup (\Sigma_i \cap \bigcup_{j \neq i}^{I} \Sigma_j))^*$. All the above projections, as well as the correspondent inverse projections, are defined also for languages in the usual manner.

2.2.3 Abstraction by decomposition

For model checking and verification of DES the state explosion problem is one of the main limitations for their application in industry. The most important technique to deal with the state explosion problem is abstraction. The basic idea of abstraction is that some parts of a given system *do not have effect* on particular properties, and hence, these parts can be *eliminated* from the system's design.

One of the most effective abstraction techniques, compositional reasoning [22], deals with complex systems in order to reduce its complexity by breaking into smaller parts, checking its properties and then deducing the system correctness. These parts do not necessary reflect the real structure of a system if any. On the other hand, the most of systems are already structured in reality. Their structure is understood by humans, the properties of their parts can be described and checked easier relatively to the complex system. Then the models of the system's parts along with their properties can be reused.

The modeling approach, described in this chapter, exploits automata framework and, which makes the idea of abstraction applicable for it.

2.3 Diagnosability of DES

In Discrete-Event Systems, diagnosis of a fault is a problem of deciding whether or not the fault occurred, under partial observation of the system's events. The problem was defined in [48] where the authors consider a monolithic model of a system. This definition and new definitions of diagnosability problem, emerged later, may be grouped as, for example, in [55], where an approach can be centralized as in [48], [39] and [61], decentralized as in [30], [44] and [45], or distributed as in [54]. The centralized and decentralized approaches require building of a monolithic model. This process corresponds with exponential growth of the model's state space, which makes the diagnosability problem intractable for complex large systems with high modularity.

Architectures for on-line diagnosis can be categorized as follows: centralized, decentralized and distributed.

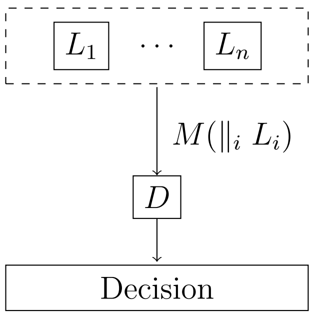


Figure 2-8: Architecture of a system with centralized diagnosis

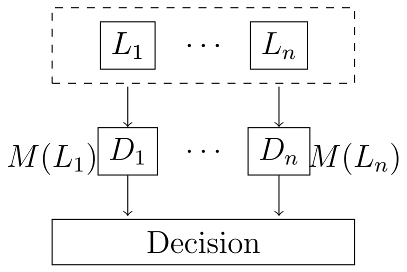


Figure 2-9: Architecture of a system with decentralized diagnosis

2.3.1 Centralized diagnosability

This architecture refers to a global model (language).If the system is modular, then the global language is built by the parallel composition of the local languages. All the observations are performed at one site. In this architecture only one diagnoser D

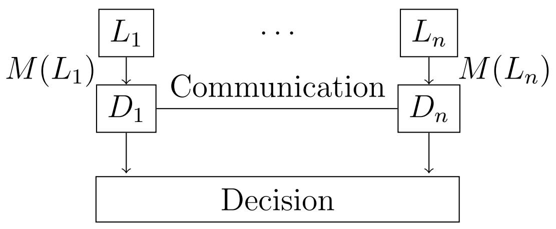


Figure 2-10: Architecture of a system with distributed diagnosis

[48] is constructed. Upon the current state of the diagnoser a decision on the fault occurrence is made. The structure is depicted in Figure 2-8.

2.3.2 Decentralized diagnosability

This approach also exploits the entire model built from its modules, but several local sites perform observations using only local diagnosers. The diagnosers do not communicate to each other, but they provide necessary information (via a protocol) to a central decision node. This architecture is depicted in Figure 2-9.

2.3.3 Distributed diagnosability

The architecture is depicted in Figure 2-10. The distributed approach does not require to built the entire model of the system. The architecture implies that the system has a set of observation spots, and each spot observes only one module of the system. A communication among observation spots is possible in order to make a decision about a fault occurrence.

The notion of modular diagnosability meets the same architectural implications, and we refer to it as to the distributed approach when the amount of information the observation spots communicate to each other is equal to zero.

Distributed approach does not require to built an entire model of the system. The approach implies that the system has a set of observation spots, carrying diagnosability information (diagnosers), each observing only a part of the system (a subset of the modules, composed). Diagnosers can communicate to each other in order to decide if a fault occurred. In general, the bigger the observation spots for the same system, the less communications among the spots is required. Figure 2-11 shows that the average size of a composed module grows while the system's modules are composed together (we imply that each module is represented by automata, and the size of a module is reflected by the number of states of a correspondent automaton). The shape of this curve depends mostly on the order the modules are taken for the composition, but the initial point, when the system has maximal modularity, and the final point, when all the modules are composed into one monolithic model, remain the same. Assuming that the system is diagnosable, the average size of a diagnoser grows much faster, since it is exponential with respect to the average size of a module, in the worst case. The amount of events the diagnosers have to exchange among each other, in order to decide if a fault occurred, is decreasing. The reason the diagnosers require less event for decision making is due to the fact that they become more self-sufficient, since amount of observable events available for each diagnoser without communications is growing. Thus, the events exchange rate decreases down to the point where no communications is required to make a decision about the faults occurrence. That is the point when the system becomes modular diagnosable (the notion of modular diagnosability was introduced in [29]). The following composition of the modules does not affect diagnosability, and leads only to the growth of the modules' and diagnosers' sizes.

In our approach, given a system with an initial natural modularity, we search for a point, when the system is modular diagnosable. If the system is diagnosable, then this point can always be found. The modules correspondent to that modularity we call *virtual*, since they differ from natural modules and used only to build diagnosers.

2.3.4 Diagnosability of a Modular System

Diagnosability analysis uses a notion of a faulty language to describe the faulty behaviour of a discrete-event system. This section discuses design issues related to representations of the faulty language and focuses on a definition of modular diagnosability.

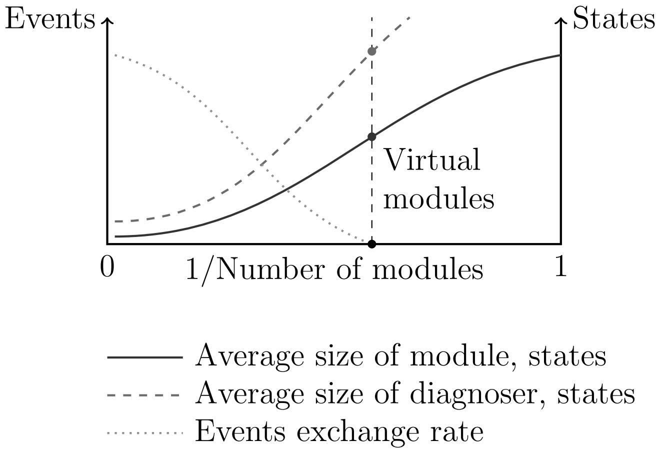


Figure 2-11: Size of modules, diagnosers, and events exchange rate, changing with respect to the degree of modularity of the system

The faulty behavior is usually modeled by introducing fault events or by faulty specifications. We refer to this approaches as to *event-based* and *specification-based* correspondingly. All the aforementioned works exploit the event-based approach, whereas the works [62] and [49] are examples of the specification-based one.

In the event-based approach fault events are a special type of event such that Σ_{uo} can be disjointed into the sets of faults Σ_f and non-faults $\Sigma_{uo} \backslash \Sigma_f$. A string containing a fault event is called *faulty string*. A set of faulty strings is called *faulty language*, i.e. formally

$$L_f := \{s \in L \mid \sigma \in s, \sigma \in \Sigma_f\}.$$

By definition, the faulty language is not necessarily prefix-closed, $L_f \subseteq \overline{L_f}$. Thus, in the event-based approach the language of the system can be partitioned into faulty and non-faulty languages, where the *non-faulty language* is defined as $L_{nf} := L \backslash L_f$.

In the case of the specification-based approach the faulty specification allows us to define undesired behavior when the fault events are not necessarily introduced. In this case this behaviour can be represented by a marked language $L_f := L_m \subseteq L$. Labeling automata's states for the same purpose can be considered as an equal technique.

Different types of undesired behaviours (or types of faults) are defined by partitioning Σ_f into subsets (not necessarily disjoint) or by several faulty specifications for the same language.

A faulty language defined in the event-based approach can be simply converted into a faulty specification by marking faulty strings, and erasing fault events. Then, we can assume that if fault events are defined, then faulty specifications can also be defined. Consequently, a set of different types of faults requires a correspondent set of specifications. Thus, a method suitable for the specification-based approach implies that it can be adopted for the event-based approach. In this paper, for the sake of unification, we use specification-based approach. For this reason the definitions of diagnosability originally developed by their authors for the event-based approach are slightly modified with no loss of meaning.

For the sake of simplicity, in the following we assume that there is only one type of fault, and that the language of the system is live.

We define *diagnosability of a fault* as follows:

Definition 1. *Given a system's language L with a fault defined by the sublanguage L_f. The fault is diagnosable if there is no two strings in the language L with the same observation such that one string is faulty and of arbitrary cardinality, and another is non-faulty, i.e. if the following holds:*

$$\begin{aligned}&\forall(s \in L_f, t \in L_f/s)\\&(\exists n \in \mathbb{N})(|t| \geq n)\\&[M(st) \cap M(L_{nf}) = \emptyset]\,.\end{aligned} \tag{2.1}$$

We define *diagnosability property of a language* as follows:

Definition 2. *The language is diagnosable if all its faults are diagnosable.*

The two above definitions altogether are similar to the Definition 1 in [48]. We recall the statement in [29] proved by Theorem 2, that the global language of the system is not diagnosable only if exists at least one non-diagnosable local language. If all the local languages are diagnosable then the global language is diagnosable. We refer to this property as to a local diagnosability property:

Definition 3 (Local diagnosability)**.** *Given the set of languages $\{L_{i\in I}\}$. The global* *language $L :=\| L_i$ is diagnosable locally if each local language L_i is diagnosable, i.e. if the following holds:*

$$\begin{aligned}
&\forall(i \in I, s \in L_{i,f}, t \in L_{i,f}/s)\\
&(\exists n \in \mathbb{N})(|t| \geq n)\\
&[M_i(st) \cap M_i(L_{i,nf}) = \emptyset]\,.
\end{aligned} \tag{2.2}$$

The definition of modular diagnosability extends the definition of local diagnosability as it takes into account the case when a faulty string locally indistinguishable in one module becomes distinguishable due to the composition with another module:

Definition 4 (Modular diagnosability)**.** *Given the set of local languages $\{L_{i\in I}\}$ and its correspondent sets $\{L_{i,f}\}$ and $\{L_{i,nf}\}$. The global language $L :=\| L_i$ is* modularly diagnosable *with respect to $M_i : \Sigma^* \to \Sigma_{i,o}^*$ if the following holds:*

$$\begin{aligned}
&\forall(i \in I, s \in L_{i,f}, t \in L_{i,f}/s)\\
&(\exists n \in \mathbb{N})(|t| \geq n)\\
&\left[M_i(P_i^{-1}(st)) \cap M_i(P_i^{-1}(L_{i,nf})) = \emptyset\right].
\end{aligned} \tag{2.3}$$

It was proved in [29] by Theorem 2, Part 2 that the local diagnosability implies the modular diagnosability[2], i.e.

$$\begin{aligned}
&\forall(i \in I, s \in L_{i,f}, t \in L_{i,f}/s)\\
&(\exists n \in \mathbb{N})(|t| \geq n)\\
&[(M_i(st) \cap M_i(L_{i,nf}) = \emptyset) \Rightarrow\\
&\left(M_i(P_i^{-1}(st)) \cap M_i(P_i^{-1}(L_{i,nf})) = \emptyset\right)]\,.
\end{aligned} \tag{2.4}$$

Recall the Definition 1 of the diagnosable fault. If a module is not diagnosable locally then exist at least two strings in its language, one is faulty and the other one is not, with the same observation of arbitrary length, i.e. the strings are *not distinguishable*. The indistinguishability can disappear if and only if: *a*) at least one string is

[2]In [62] the authors show that the local diagnosability and modular diagnosability are not comparable but they have a different setup for the problem.

not in its language due to concurrency with other module, and then the strings would be distinguishable locally - the verification of the modular diagnosability property is devoted to find if this is the case; *b*) indistinguishability is broken globally by interleaving sequences of the module's events with observable events of other modules. The later case is expressed in the following conjecture:

Conjecture 1. *Given a system of two modules with languages L_1 and L_2, and the global language $L := L_1 \parallel L_2$. Suppose there is only one faulty string $s \in L_1$ such that it is not distinguishable from at least one string of $L_1 \backslash s$. Thus, L_1 is not locally diagnosable. Suppose the system is not modular diagnosable. Then the global language L is diagnosable only if all the strings $t \in P_1^{-1}(s)$ change their observation due to the composition with the language L_2.*

The above conjecture gives the insight into the underlining idea of our approach. If we find a module which makes the faulty string distinguishable then the composition of that module with a faulty one would result in a new module satisfying the property of local diagnosability, thus improving the modular diagnosability property of the system. In the following section we provide a formal description of the problem.

Chapter 3

Virtual Modular Diagnosability

This chapter presents the virtual modular diagnosability, a new type of diagnosability. After a formal definition of the notion, it proceeds with an analysis of a simple system which consists of two modules, modeled by automata. This analysis exposes structural conditions for the models, such that a verification whether the models satisfy these conditions allows to infer diagnosability property of their composition, i.e of the entire system. Then these conditions are rewritten for a general case, i.e. a system that consists of an arbitrary number of components. The chapter then presents algorithms which facilitate diagnosability verification in a efficient way. At the end, a simple abstract example shows application of this approach.

The final goal is to make a given system modularly diagnosable. Lets assume that the system is diagnosable globally with respect to a given fault. It implies that it is always possible to find a subset of the system's modules, such that it contains a faulty module, and all the modules from the subset can be composed into one locally diagnosable module. Each module in the subset helps to detect and distinguish the fault by sequences of its observable events. The assumption also implies that not each module of the system can or may participate in the fault detection and distinguishing process.

3.1 Formal Definition

Mathematically speaking, if the initial modularity does not satisfy the property of modular diagnosability, then we need to find a partition of the set of the modules such that all the modules in each element of the partition can be considered as a *virtual module*, and the system with the new modularity satisfies the property of modular diagnosability.

The setup for the problem is following. Let a system be defined as a set of local languages $\{L_{i\in I}\}$, and let a faulty behaviour be defined for any $i \in I$, i.e. each language L_i can be disjoint into faulty and non-faulty sublanguages: $L_{i,f}$ and $L_{i,nf}$. For the sake of simplicity, we assume that a language may have only one type of fault. We assume that $(\forall i \in I)\,[L_i = P_i(L)]$. The motivation for this assumption is following. Most of real-life complex systems are of high modularity, where each module is presented as a small model along with correspondent faults as, for example, in [49]. When properly designed, such systems have neither deadlocks nor trajectories which never executed. In practice, such assumption may require construction of a supervisor, and verification that a local language of each module is globally consistent, before checking diagnosability of the system.

Definition 5. *Let J be a partition of I. Given a set of system's languages $\{L_{i\in I}\}$ and a set of observable projections $\{M_j \mid j \in J\}$. The system is modularly diagnosable with respect to J and $\{M_j\}$ if the following holds:*

$$\begin{aligned}
&\forall(i \in I, s \in L_{i,f}, t \in L_{i,f}/s)\\
&(\exists n \in \mathbb{N})(|t| \geq n)\\
&\left[M_j(P_i^{-1}(st)) \cap M_j(P_i^{-1}(L_{i,nf})) = \emptyset\right].
\end{aligned} \tag{3.1}$$

The definition requires each fault originated in any language of the subset j to be diagnosed by observing only events of the languages from the same subset j.

A trivial solution for the above problem is to enumerate all the partitions of the set and check whether the result of composition of each element of the partition is locally diagnosable. In general, the total number of partitions of a set with cardinality n is known as the Bell number B_n, which has a double exponential generation function [25].

In reality, it may not necessary to enumerate all the partitions, since a partition which corresponds to modularly diagnosable system can be found even at the first iteration. Moreover, it is not necessary to compose all the modules of each element of the partitions, since not all partition's elements may contain modules with faults. However, the level of complexity in the worst case motivates us to find a more smart solution for the problem.

In the next section we make the first step toward a feasible approach for the problem.

3.2 Trivial system. Analysis of two adjacent modules

As was mentioned above, the first step to improve the search for a relevant partition, is to enumerate only the modules which potentially can influence diagnosability. Firstly, those are ones which have observable events. Then, the structure of modules may have some patterns, such that if a pattern is found, then we can infer its influence before the module is composed with others. The module with the fault may also be analyzed for a possibility to change its observation due to interaction with other modules. Thus, in the trivial case of a system consisting of two modules where only one module has a fault, in order to solve the problem, we need to answer the following questions: a) can a module with a fault potentially change observations of its strings while composed with the adjacent module? and b) can a module change observation of some strings of another module due to concurrency? No parallel composition should be used in order to answer the questions. Otherwise, the problem would reduce to the problem of local diagnosability.

We suppose that the system consists only of two modules with the correspondent languages L_1 and L_2. The language of the system is $L := L_1 \parallel L_2$. Suppose that only one module has a faulty behaviour: $L_1 := L_{1,f} \dot\cup L_{1,nf}$. Suppose that L_1 is not diagnosable locally, but L is diagnosable.

Firstly, we define the notion of observation changing of a string in a global language.

Definition 6. *Given two languages L_1 and L_2. A string $s \in L_1$* changes its observation *$M_1(s)$ in the language L if there is no the same observation in $P_1^{-1}(s)$, i.e. if the following holds:*

$$M(L) \cap M_1(s) = \emptyset. \tag{3.2}$$

Lemma 1. *Given two languages L_1 and L_2, and a string $s \in L_1$. Assume that $s \in P_1(L)$. The string s changes its observation in the language L if and only if:*

$$(\exists \sigma \in s \mid \sigma \in \Sigma_1 \cap \Sigma_2) \wedge \tag{3.3a}$$

$$(\forall t\sigma \in L_2)\,[M_2(t) \neq \emptyset]\,, \tag{3.3b}$$

$$\text{where } M_2 : \Sigma^* \to (\Sigma_{2,o} \backslash \Sigma_1)^*. \tag{3.3c}$$

Proof. In order to prove sufficiency of (3.3) we use its converse relation and prove by contradiction that the change of observation (3.2) is necessary. Assume $\exists w \in L$ and $\exists s \in L_1$ such that $M(w) = M_1(s)$ and, therefor, (3.2) is false. Let $\exists \sigma \in \Sigma_1 \cap \Sigma_2$ such that $\sigma \in w$ and also (3.3a) holds. Then may $\exists u\sigma \in \overline{w}$ such that $M_2(u) = \emptyset$, and then $M_2(P_2(u)) = \emptyset$ which contradicts (3.3b). Now, let (3.3b) be true for all $t\sigma \in P_2(w)$. Then the assumption $M(w) = M_1(s)$ holds only if $\sigma \notin s$, which contradicts (3.3a).

We prove necessity of (3.3) by contradiction. Let (3.3a) holds, and $\exists t\sigma \in L_2$ such that $M_2(t) = \emptyset$. Then may $\exists t'\sigma \in L \subseteq P_2^{-1}(t\sigma)$ such that $M_2(t') = \emptyset$ and $M_1(t') = M_1(s) \neq \emptyset$, which contradicts (3.2). Now, let (3.3b) holds and $\nexists \sigma \in s' \in P_1^{-1}(s) \mid \sigma \in \Sigma_1 \cap \Sigma_2$. Then may $\exists w \in L_2$ and, hence, $w' \in L \subseteq P_2^{-1}(w)$ such that $M(w') = M(s)$, which contradicts (3.2). $\square$

Informally, the above lemma says that the string of the local language L_1 changes its observation in the global language L if and only if the string has an event in common with the language L_2, and all the strings of L_2 which have this common event have observable events in the prefixes, and some of the observable events in the prefixes are not common with L_1.

We call the subset of stings $\{t \in L_2\}$ satisfying condition (3.3b) as the *adjacent observable support* for the given string $s \in L_1$.

Definition 7. *Given two languages L_1 and L_2. We say that a string $s \in L_1$ is distinguished from all the other local strings $L_1 \backslash s$ in the language L if the following holds:*

$$\begin{aligned}&(\forall w \in L_1 \backslash s)\\ &\left[MP_1^{-1}(w \parallel L_2) \cap MP_1^{-1}(s \parallel L_2) = \emptyset\right].\end{aligned} \tag{3.4}$$

Lemma 2. *Given two languages L_1 and L_2. Assume that $L_1 = P_1(L)$. The string $s \in L_1$ is distinguished from $L_1 \backslash s$ in the language L if s has an adjacent observable support $L_{2,s} \subseteq L_2$ which satisfies the following condition:*

$$(\forall t \in L_{2,s})\,[\exists \sigma \in t \mid \sigma \in \Sigma_1 \cap \Sigma_2)] \wedge \tag{3.5a}$$

$$(\forall w \in L_1 \backslash s)\,[\sigma \notin w] \wedge \tag{3.5b}$$

$$(\forall t'\sigma \in \bar{t})[M_2(t/t'\sigma) \neq \emptyset], \tag{3.5c}$$

where M_2 is defined as in (3.3c).

Proof. Assume (3.4) is false, i.e. $\exists w' \in P_1^{-1}(w)$ and $\exists s' \in P_1^{-1}(s)$ such that $M(w') = M(s')$.

Assume (3.5a) and (3.5b) hold. Then may $\exists t' \in \overline{s'} \mid t' \in P_2^{-1}(L_{2,s})$ such that $M(t') = M(w')$, and $t \in P_2(t')$ such that $M_2(t) = M_2(w)$. And may $\exists t'' \in \bar{t}$ such that $M_2(t'') = M_2(w)$. Since $\sigma \in t$ and $\sigma \notin w$, then $M(t \backslash t''\sigma) = \emptyset$ for any $t''\sigma \in \bar{t}$, which contradicts (3.5c).

Assume (3.5a) and (3.5c) hold. Let $M_1(L_1) = \emptyset$ and $M_2(t \backslash t'\sigma) = M_2(s) = M_2(s)$. Then $\forall s' \in M(P_1^{-1}(s))$ there exists $\sigma \in s'$, which contradicts (3.5b).

Assume (3.5b) and (3.5c) hold. If (3.4) is false, then (3.5a) is false. However, (3.5c) is sufficient for (3.5a), which contradicts the former statement. $\square$

Informally, the above lemma says that a string s becomes distinguishable from the other strings $L_1 \backslash s$ in the global language, when the occurrence of events from the observable support happens only in $P^{-1}(s)$ due to common events. Thus, whenever we observe events of the observable support of L_2, we are sure the string s in L_1 is being executed.

Indistinguishability can be changed either by blocking the string in the local language due to concurrency, or by interleaving with observable events from other languages. Under assumption that all the strings are not affected by concurrency, i.e. $L_1 = P_1(L)$ we can deduce, that the conditions of Lemma 4 are also necessary for changing distinguishability.

The Figure 3-1 depicts an automaton which accepts the sublanguage of L_2 satisfying conditions (3.3a), (3.5a) and (3.5c) of the above lemmas. The Figure 3-2 depicts an automaton which marks a sublanguage of L_1 satisfying conditions (3.3a) and (3.5a).

A procedure verifying if a string $s \in L_1$ is distinguishable in the global language L consists of two steps. First, the string s should be marked by the automaton depicted in the Figure 3-2. Then the set of common events $\Sigma_1 \cap \Sigma_2$ is reduced to the set of events causing transitions in the automaton. Second, all the continuations of the strings of the language L_2 which have these common events should be accepted by the automaton depicted in the Figure 3-1.

Now we are ready to apply Lemma 4 with respect to diagnosability property, but make some notes before. Intuitively, one would apply the conditions of the lemma for faulty and non-faulty languages. Recall, that faulty and non-faulty languages are disjoint, but they may have common prefixes. This common sublanguage is defined as $\overline{L_f} \cap (L \backslash \overline{L_f})$. Changing observability of this sublanguage has no effect for diagnosability, and we can exclude it from a verification procedure. Thus, the

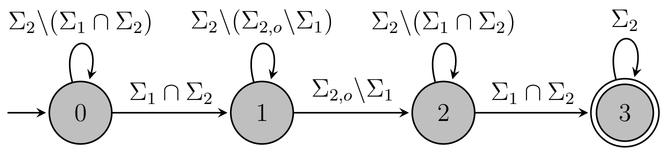

$\Sigma_2\backslash(\Sigma_1\cap\Sigma_2)$
$\Sigma_2\backslash(\Sigma_{2,o}\backslash\Sigma_1)$
$\Sigma_2\backslash(\Sigma_1\cap\Sigma_2)$
Σ_2
0
$\Sigma_1\cap\Sigma_2$
1
$\Sigma_{2,o}\backslash\Sigma_1$
2
$\Sigma_1\cap\Sigma_2$
3

Figure 3-1: Automaton for marking the language L_2

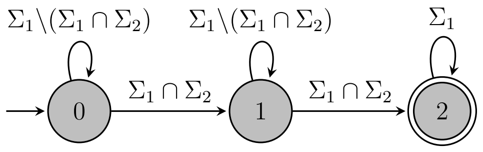

$\Sigma_1\backslash(\Sigma_1\cap\Sigma_2)$
$\Sigma_1\backslash(\Sigma_1\cap\Sigma_2)$
Σ_1
0
$\Sigma_1\cap\Sigma_2$
1
$\Sigma_1\cap\Sigma_2$
2

Figure 3-2: Automaton for marking the language L_1

non-faulty sublanguage disjoint to all the prefixes of the faulty language is defined as $L\backslash\overline{L_f}$, and the set of all prefixes of the faulty language disjoint to the above non-faulty sublanguage and to the common prefixes is defined as $\overline{L_f}\backslash(\overline{L_f}\cap(L\backslash\overline{L_f}))$.

Lemma 3. *Given L_1, L_2, $L_{1,f}\subseteq L_1$ and $L_{1,nf}\subseteq L_1$. A language $L := L_1 \parallel L_2$ is diagnosable if the sublanguages $\overline{L_{1,f}}\backslash(\overline{L_{1,f}}\cap(L_i\backslash\overline{L_{1,f}}))$ and $L_1\backslash\overline{L_{1,f}}$ have distinguished observable supports in L_2.*

The proof can be deduced from the Lemma 4.

The automata depicted in Figures 3-1 and 3-2 can not be simply used in a procedure verifying diagnosability, since we should avoid the verification of $\overline{L_{1,f}}\backslash L_{1,f}$ and $L_{1,nf}\backslash\overline{L_{1,f}}$. However, it can be used to demonstrate the approach in a trivial case, as it is shown in the next section.

3.2.1 Example

Consider the system of two automata G_1 and G_2 depicted in Figure 3-3 and Figure 3-4. The set of events for the system is $\Sigma = \{a, b, c, e, f\}$. Suppose the observable events are $\Sigma_o = \{c, e\}$, and the set of fault events is $\{f\}$. Thus, only the language L_1 has a fault, and L_2 has not. We use the verifier [61] to check if a language is diagnosable. The verifier for the language L_1 is depicted in the Figure 3-5. One can check that it has an indeterminate cycle. The strings fbc^* and ac^* are not distinguishable in the

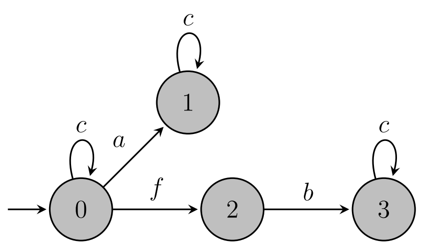


Figure 3-3: Automaton G_1

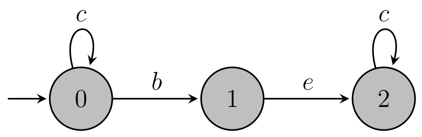


Figure 3-4: Automaton G_2

local language L_1. Hence, L_1 is not locally diagnosable.

We now use the verification procedure described in this paper to check if the language L_2 can changes observation of either strings fbc^* or ac^* in the language $L_1 \parallel L_2$ such that the strings become distinguishable. The set of events common for L_1 and L_2 is $\{b, c\}$. It can be verified that only the strings fbc^* are marked by the automaton depicted in the Figure 3-2. Next, it can be verified that all the strings of L_2 which have events common with the strings fbc^* are accepted by the automaton depicted in the Figure 3-1. Thus, we conclude that L_2 changes observation of the strings fbc^* in the virtual module G built of modules G_1 and G_2, such that G becomes diagnosable. Indeed, if we make a parallel composition of the modules and build a verifier for the result as it is depicted in the Figure 3-6, it can be checked that the verifier has no indeterminate cycles.

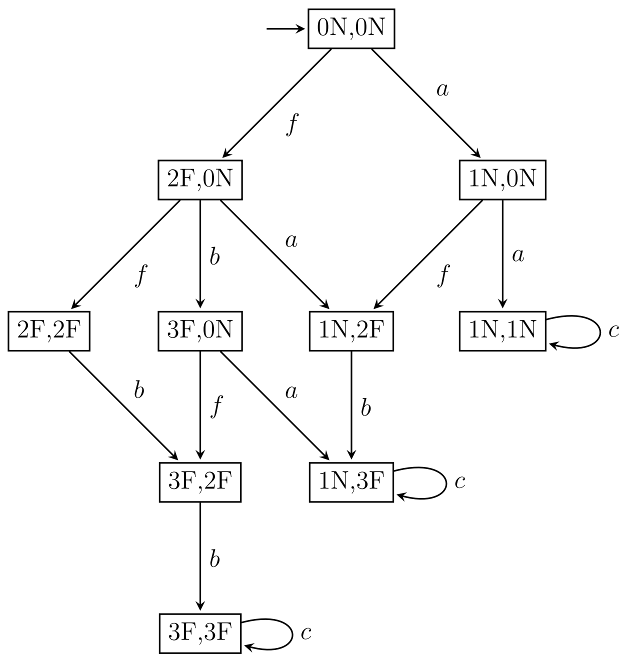


Figure 3-5: Verifier of G_1

3.3 Generalization. Conditions for diagnosability

We rewrite Lemma 2, where the statements hold for a system of two languages, for an arbitrary couple of languages as follows.

Lemma 4. *Given a tuple* $\langle L_i, L_j, s, u\rangle \mid i, j \in \mathbb{N}, i \neq j$ *and* $s, u \in L_i$*. The strings* s, u *are distinguished in the language* $L_i \parallel L_j$ *if for the string* s *exists an adjacent observable support* $K_j \subseteq L_j$ *which satisfies the condition:*

$$\begin{aligned}&(\forall t \in K_j)(\exists \sigma \in t \mid \sigma \in \Sigma_i \cap \Sigma_j)(\sigma \notin u)\\&(\forall w\sigma \in \bar{t})[M_j(t/w\sigma) \neq \emptyset].\end{aligned} \tag{3.6}$$

Informally, the lemma imposes a requirement for the string s to have an *observable support* (subset of observable strings ending with common events) in another language, which guaranties that we can observe some events in that language before any continuation of the string s. Then, the lemma defines a condition which allows us to distinguish these observations from the observation of the string u.

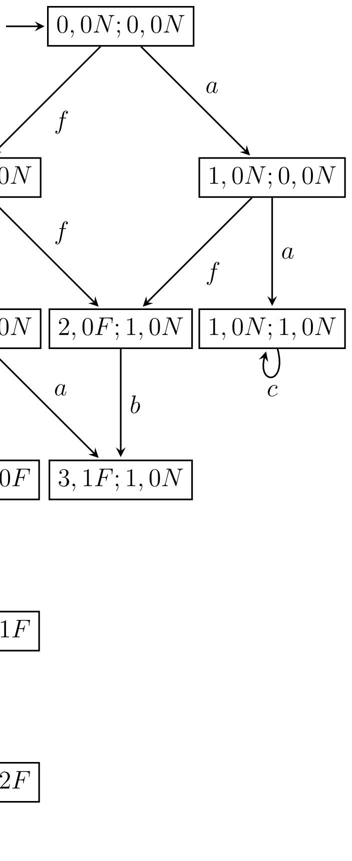


Figure 3-6: Verifier of $G_1 \parallel G_2$

Now, we rewrite Lemma 3, in the context of our setup, as follows:

Lemma 5. *Given a virtual module $j \in J$, the module is diagnosable with respect to M_j if:*

$$\begin{aligned}
&(\forall i \in j, s \in P_i^{-1}(L_{i,f}), t \in P_i^{-1}(L_{i,nf})) \\
&(\exists k \in j \mid k \neq i) \text{ and} \\
&\text{condition (3.6) holds for at least one of the tuples:} \\
&\left\langle P_i^{-1}(L_i), L_k, s, t \right\rangle, \left\langle P_i^{-1}(L_i), L_k, t, s \right\rangle.
\end{aligned} \tag{3.7}$$

The lemma implies, that the virtual module is diagnosable if we can distinguish all faulty strings from all non-faulty strings. From the lemma it can be seen that for one faulty sublanguage there can be a subset of languages with observable supports (a few $k \in j$), satisfying the conditions. In order to make the system modularly diagnosable, thus, one needs to find a relevant partition, with a desired distribution of modules over the partition. It worth to note, that the partitioning implies that languages with faults are in subsets with the languages which have observable events. Thus, other languages which have no faults and have no observable events, as well the languages with observations which have no effect for the diagnosability property, will be in other subsets.

One faulty module may require a few modules with observable supports. The algorithms presented in the following section are aimed to find such subset of system's modules. The first algorithm may be viewed as a *forward propagation* procedure. It starts from the faulty module, and propagates faulty and non-faulty information trough all the modules, according to some of requirements of the lemmas 4 and 5 related to common events. The second algorithm may be viewed as a *backward propagation* procedure. It starts from a module with observable support, and propagates information related to observable events toward the faulty module, according to the rest of the lemmas' requirements.

3.4 Algorithms for diagnosability by virtual modules

Assume that a language with a faulty behaviour and a language which has an observable support are given. Then, it's required to verify if the observable support is either for the faulty or non-faulty sublanguages. If it is true, then the correspondent modules can be composed into a virtual module. According to the Lemma 5, in order to decide if the virtual module is diagnosable we need to verify the necessary conditions for all the strings $P_i^{-1}(L_{i,f})$ and $P_i^{-1}(L_{i,nf})$, projected to the local languages of other modules $\{j \in I \mid j \neq i\}$. Those local projections are the basic elements our algorithms operate with. In the sequel we refer to the correspondent local sublanguages as to *co-faulty* and *co-non-faulty* ones. A formal definition of these sublanguages is presented below, and then an algorithm which computes the co-faulty and co-non-faulty sublanguages for all the system's modules, is given.

3.4.1 Co-faulty and co-non-faulty sublanguages

Definition 8. *Given $i, j \in I$, $L_{i,f} \subseteq L_i$, we say that a sublanguage $L_{j,cf} \subseteq L_j$ is* co-faulty *with respect to $L_{i,f}$ if*

$$(\forall s \in L_{j,cf})(\forall t \in P_i[P_j^{-1}(s)])\, [t \in L_{i,f} \wedge t \notin L_{i,nf}]\,,$$

and a sublanguage $L_{j,cnf} \subseteq L_j$ is co-non-faulty *with respect to $L_{i,nf}$ if*

$$(\forall s \in L_{j,cnf})(\forall t \in P_i[P_j^{-1}(s)])\, [t \notin L_{i,f} \wedge t \in L_{i,nf}]\,.$$

In words, the co-faulty sublanguage is a sublanguage which satisfies two conditions: a) it is the sublanguage of projection of the global faulty language to the local language; b) the sublanguage is not co-non-faulty. Similarly, the co-non-faulty sublanguage is a sublanguage which satisfies conditions: a) it is the sublanguage of projection of the global non-faulty language to the local language; b) the sublanguage is not co-faulty.

Given global faulty and non-faulty sublanguages, one can compute the local co-faulty sublanguage, using the global language, as follows:

$$\begin{aligned} L_{j,cf} &:= P_j(L_f)\backslash P_j(L_{nf}), \\ L_{j,cnf} &:= P_j(L_{nf})\backslash P_j(L_f). \end{aligned} \tag{3.8}$$

However, we are interested to compute local sub languages without computing the global language. Since the global faulty sublanguage is define as

$$L_f := P_i^{-1}(L_{i,f}) \cap \bigcap_{k\neq i}^{I} P_k^{-1}(L_k), \tag{3.9}$$

then, by substituting the global language in 3.8 by 3.9, and by mathematical induction, the co-faulty sublanguage of an arbitrary module can be computed as follows:

$$\begin{aligned}
&L_{j,cf} := \\
&P_j\Big(P_j^{-1}(L_j) \cap P_i^{-1}(L_{i,f}) \cap \textstyle\bigcap_k^I P_k^{-1}(L_{k,cf})\Big)\setminus \\
&P_j\Big(P_j^{-1}(L_j) \cap P_i^{-1}(L_{i,nf}) \cap \textstyle\bigcap_k^I P_k^{-1}(L_{k,cnf})\Big), \\
&\text{where } i \neq j \neq k \in I.
\end{aligned} \tag{3.10}$$

We skip the definition of co-non-faulty sublanguage here due to its similarity.

3.4.2 Fault propagation algorithm

Now, we present Algorithm 1 which, given a module with a faulty behaviour defined, computes the co-faulty and co-non-faulty languages for all the rest modules of the system. The algorithm starts from a preliminary step at line 1. At this step a projection to the common events C_j is computed for all the local languages. Projections of co-faulty and co-non-faulty sublanguages to the common events, denoted as F_j, N_j, are empty in the beginning. Event sets $\Sigma_{j,c}$ are sets of the common events, which don't belong neither to co-faulty and co-non-faulty sublanguages nor to intersection of the sublanguages' prefix-closures. Since that sublanguages are not defined in the beginning, the event sets contain all events from the projections. A role of these sets will be clarified later. The recursive procedure *propagate-fn* computes F_j and N_j for all the system's modules, except the module with a fault; the procedure operates with common events only. Before entering the procedure the projections of the faulty module's sublanguages, F_i, N_i are computed at the step 22. Note, that projection operation preserves faulty and non-faulty information, when faulty behaviour is defined in a specification-based way.

The recursive procedure computes all projections F_j, N_j of the modules adjacent to a given module k (we say that two modules are adjacent to each other if their languages have events in common). At the first steps 8-12 of the procedure we compute F'_j, N'_j with respect to the module k; these projections are partial. The parallel composition $F_k \parallel C_j$, followed by projection to the common events, is used to extract

```
Require: The set of the system's languages $\{L_j \mid j \in I\}$, faulty and non-faulty
    sublanguages $L_{i,f}$, $L_{i,nf}$ of a faulty module $i \in I$.
 1: for all $j \in I$ do                                          ▷ Initialization
 2:     $\Sigma_{j,c} \leftarrow \Sigma_j \cap \bigcup_{i \neq j}^{I} \Sigma_i$
 3:     $C_j \leftarrow P_{j,c}(L_j)$
 4:     $F_j \leftarrow \emptyset, N_j \leftarrow \emptyset$
 5: end for
 6: procedure PROPAGATE-FN($k$, $F_k$, $N_k$)
 7:     for all $j \in I \mid j \neq i, \Sigma_j \cap \Sigma_k \neq \emptyset, \Sigma_{j,c} \neq \emptyset$ do
 8:         $F'_j \leftarrow P_{j,c}(F_k \parallel C_j)$
 9:         $N'_j \leftarrow P_{j,c}(N_k \parallel C_j)$
10:         $K \leftarrow \overline{F'_j \cap N'_j}$
11:         $F'_j \leftarrow F'_j \backslash K$
12:         $N'_j \leftarrow N'_j \backslash K$
13:         if $F'_j \backslash F_j \neq \emptyset \vee N'_j \backslash N_j \neq \emptyset$ then
14:             $F_j \leftarrow F_j \cup F'_j$
15:             $N_j \leftarrow N_j \cup N'_j$
16:             $\Sigma_{j,c} \leftarrow \{\sigma \in s\}$, where
17:             $s \in \left[(\overline{F_j} \backslash F_j) \cup (\overline{N_j} \backslash N_j)\right] \backslash (\overline{F_j} \cap \overline{N_j})$
18:             PROPAGATE-FN($j, F_j, N_j$)
19:         end if
20:     end for
21: end procedure
22: $F_i \leftarrow P_{i,c}(L_{i,f})$
23: $N_i \leftarrow P_{i,c}(L_{i,nf})$
24: PROPAGATE-FN($i$, $F_i$, $N_i$)                              ▷ Beginning of recursion
25: for all $j \in I \mid j \neq i$ do                           ▷ Finalization
26:     $L_{j,cf} \leftarrow L_j \cap P_{j,c}^{-1}(F_j)$
27:     $L_{j,cnf} \leftarrow L_j \cap P_{j,c}^{-1}(N_j)$
28: end for
29: return $\{L_{j,cf}\}$, $\{L_{j,cnf}\}$
```

Algorithm 1: Forward propagation of a fault. Computes co-faulty and co-non-faulty languages of all the modules

the faulty information from the language F_k to C_j, and non-faulty from N_k to C_j. At the step 13 we check if some co-faulty or co-non-faulty strings should be added to the projections computed earlier. If so, then we update F_j, N_j, $\Sigma_{j,c}$ and call the same procedure to update projections of the adjacent modules. Thus, whenever the co-faulty or co-non-faulty projections of that module changed, the projections F_j, N_j are updating with respect to every other module, due to recursion. The event sets

```
Require: The index $i \in I$ of the faulty module, the index of a module with observable
    support $k \in I \mid M(L_{k,cf}) \neq \emptyset \vee M(L_{k,cnf}) \neq \emptyset, k \neq i$, the sets $\{L_{j,cf}\}, \{L_{j,cnf}\}, j \in I$.
 1: for all $j \in I$ do
 2:     $D_{j,f} \leftarrow P_{j,co}(L_{j,f})$
 3:     $D_{j,nf} \leftarrow P_{j,co}(L_{j,nf})$
 4: end for
 5: procedure PROPAGATE-D($k$, $D_{k,f}$, $D_{k,nf}$)
 6:     for all $j \in I \mid \Sigma_j \cap \Sigma_k \neq \emptyset$ do
 7:         $D'_{j,f} \leftarrow P_{j,co}(F_j \parallel D_{k,f})$
 8:         if $(D'_{j,f} \backslash D_{j,f} \neq \emptyset) \vee (D'_{j,nf} \backslash D_{j,nf} \neq \emptyset)$ then
 9:             $D_{j,f} \leftarrow D'_{j,f}$
10:             $D_{j,nf} \leftarrow D'_{j,nf}$
11:             PROPAGATE-D($j$, $D_{j,f}$, $D_{j,nf}$)
12:         end if
13:     end for
14: end procedure
15: PROPAGATE-D($k$, $D_{k,f}$, $D_{k,nf}$)
16: $L^{ext}_{i,f} \leftarrow P_{i,co}(L_{i,f} \parallel D_{i,f})$
17: $L^{ext}_{i,nf} \leftarrow P_{i,co}(L_{i,nf} \parallel D_{i,nf})$
18: return $L^{ext}_{i,f}, L^{ext}_{i,nf}$
```

Algorithm 2: Backward propagation of the fault observation. Computes observable information for the faulty module

$\Sigma_{j,c}$ represent a potentiality that the projections F_j, N_j can be changed; the modules which have these sets empty can't change their language's property with respect to the fault and, thus, are skipped during recursive procedure. Finally, all the co-faulty and co-non-faulty sublanguages are calculated for each module at the step 25.

3.4.3 Observation propagation algorithm

The structure of the Algorithm 2 is similar to one in Algorithm 1. It starts from computing projections to common and observable events for the co-faulty and co-non-faulty sublanguages of each module's language. Then, starting from the module k which has observable events in its sublanguages (i.e. it has potential observable support for a sublanguage of the faulty module), begins a recursive procedure at step 15. In the procedure we collect only observable events of other modules which are related to co-faulty or co-non-faulty sublanguages. Then we propagated these events down to the faulty module. Finally, at the step 16, the faulty and non-faulty sublanguages of the faulty module are extended with observable events.

The result of the Algorithm 2, the faulty and non-faulty sublanguages extended with observable events, can be used to verify diagnosability now. Any approach, which allows us to check for the presence of indistinguishable strings in the language (the faulty and non-faulty sublanguages can be united, if necessary), is suitable. We can have the following outcomes after verification: a) no indistinguishable strings are broken, b) no indistinguishable strings left, and c) some indistinguishable strings are broken. In case (a) we pick another module $k \in I$ satisfying requirements of the Algorithm 2 for further processing and verification. In case (b) we may construct a virtual module of the faulty module i and the module k and declare that the fault is diagnosable with respect to the virtual module $\{i, k\}$ and stop, or continue and check if exists another module which can also be used to build a virtual module. In case (c) we should pick additional module satisfying requirements of the Algorithm 2 and continue the process until all the indistinguishable strings are broken. The following section analyzes what may be considered as a criteria for choosing the module with observable support k.

3.4.4 Algorithm to choose a module with observable support

Assume that the objective function is to have the number of modules in the system as many as possible, i.e. to have the maximal cardinality of the partition J. Given an initial set of languages, let rank of J be equal to 0, i.e. $|J| = |I|$. Let F denotes the subset of languages with faults, and M denotes the subset of languages with correspondent observable supports. Assume that it is always possible to make the system modularly diagnosable with respect to some virtual modules. Then we know that $(\forall f \in F)\ \exists m \in M$. Let $F \cap M = \emptyset$ and $|M| \geq |F|$. Then, to make the system modularly diagnosable it is required, in the worst case, $|F|$ members of M. Thus, the cardinality of the systems with virtual modules, i.e. the cardinality of the partition, in the worst case is:

$$\min |J| = |I| - |F|. \qquad (3.11)$$

Let $M \subseteq F$. Then, the cardinality of the partition in the best case is:

$$\max |J| = |I| - \left(\left\lfloor \frac{|F|}{2} \right\rfloor + \operatorname{mod} \frac{|F|}{2} \right). \tag{3.12}$$

As can be seen from the above, in order to maximize the number of modules while making the virtual modules, we should consider, firstly, the modules which themselves have faults. We propose a procedure for the given objective function as follows:

Given the system of cardinality $|I|$, the subset $F \subseteq I$ of modules with faults, such that any $f \in F$ is not diagnosable locally, the subset $M \subseteq I$ of modules with observable supports, such that $(\forall f \in F)\ \{m \in M\} \neq \emptyset$. In order to make the system diagnosable by the minimal number of virtual modules:

1. Initially set the partition J of I such that the rank of J is equal to 0;
2. $(\forall j \in J, f \in j)$ update j as follows: $j := j \cup \{m \in M \cap F\}$ if $M \cap F \neq \emptyset$, $j := j \cup \{m \in M\}$ otherwise, where $\{m\}$ is a singleton of the arbitrary chosen observable support for the given f, such that the new j is locally diagnosable; exclude m from $k \in J \mid k \neq j$, and remove k from J if $k = \emptyset$.

The order the observable supports are verified in the procedure guaranties that we consider the languages which themselves contain the faulty sublanguages at first, thus maximizing the number of virtual modules in the system.

It worth to note that the above algorithm can be augmented by additional criteria aimed, for instance, to decrease computational burden.

3.5 Example

An example with languages presented by automata (see the Figures 3-7 – 3-12) shows how the algorithms 1 and 2 can be applied to a simple system. We use marked states to mark strings in languages, since they are not prefix-closed, in general. The system consists of three modules as depicted in Figure 3-7. Here a fault is presented only in the language L_1 by the event f. Figure 3-8 depicts faulty and non-faulty sublanguages of the language L_1. Projections of these sublanguages to the common events (step 22 of the Algorithm 1), F_1 and N_1 are depicted in the Figure 3-9. Suppose that in the beginning of the recursive procedure we pick module 2. The figure depicts the co-faulty and co-non-faulty sublanguage of the language L_2 with respect to the language L_1, calculated at the steps 8 of the algorithm (we don't show intermediate automata, reflecting compositional steps, due to space restrictions). Note, that, when calculated, the event set $\Sigma_{2,c}$ is empty for module 2. Hence this module will not participate in the procedure as a module adjacent to others. At the next step we randomly pick the language L_4, and compute F_4 and N_4 with respect to the module 2. Now, note that $\Sigma_{3,c} := \{e\}$, since this event is neither in the co-faulty and co-non-faulty projections nor in their common prefix. The iterations is followed by the module 3, which has module 4 as an adjacent one. Therefore, the sublanguages of this module are updated.

Figure 3-10 depicts co-faulty and co-non-faulty sublanguages of module 4 calculated at the step 16 of the Algorithm 1. It shows that the co-faulty sublanguage of the module has an observable support; it satisfies the requirements of the Lemma 4. Other modules with no faults have no observable events; they have no observable supports, and not depicted.

The major steps of Algorithm 2 are depicted in the Figure 3-11. It is shown how the observable information from the module 4 is propagated backward to the language of the module 1. The final step of the algorithm is depicted in the Figure 3-12. Clearly, the faulty and non-faulty sublanguages, extended with observable information, have distinguished observations. Then, we can conclude that the virtual module composed of the modules 1 and 4 is locally diagnosable. Hence, the system is modularly diagnosable with modularity presented by the partition $\{\{1, 4\}, \{2\}, \{3\}\}$.

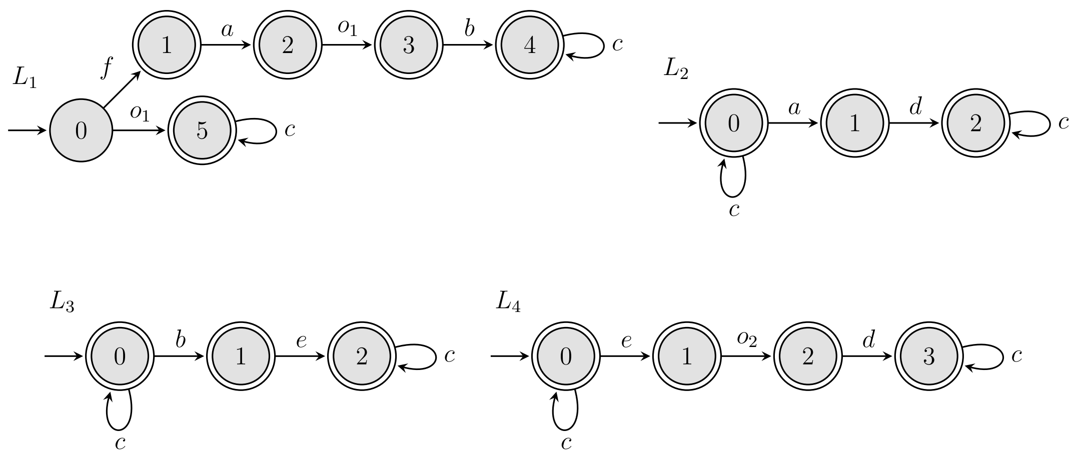


Figure 3-7: Automata marking languages L_1, L_2, L_3 and L_4. $\Sigma_o := \{o_1, o_2\}$, $\Sigma_f := \{f\}$

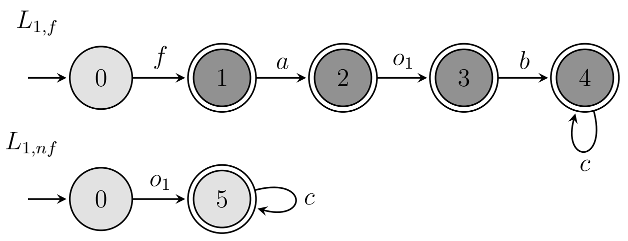


Figure 3-8: Automata marking the faulty and non-faulty sublanguages of L_1

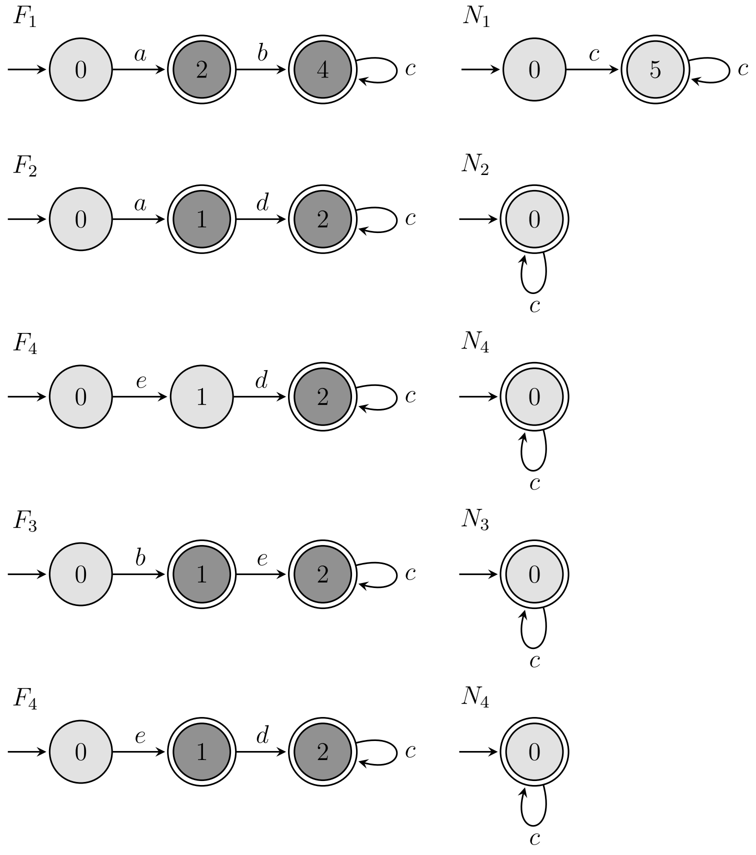


Figure 3-9: Automata marking sublanguages F_1 and N_1 of the faulty module, and co-faulty and co-non-faulty languages of the modules 2, 3 and 4 in the order they are composed (top-down): 1-2, 2-4, 1-3, 3-4. Note, that the sublanguage F_4 is defined partially at first, then fully

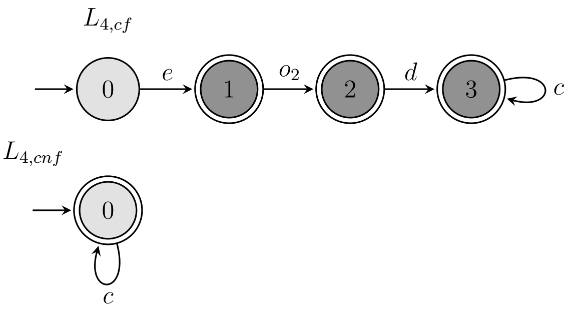


Figure 3-10: Automata marking co-faulty and co-non-faulty sublanguages of the language L_4. Automata for the sublanguages of the modules 2 and 3 are equal to the ones depicted in the Figure 3-9

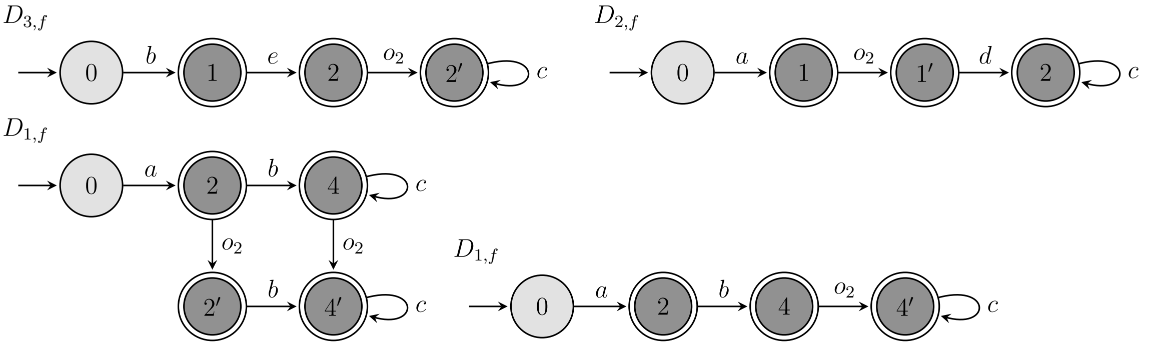


Figure 3-11: Automata marking $D_{j,f}$ of the modules 1, 2 and 3 in the order they are composed (top-down): 4-3, 4-2, 2-1, 3-1. Automata marking $D_{j,nf}$ are not depicted since they have no observable events

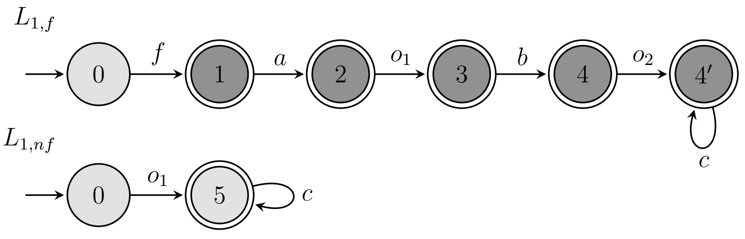


Figure 3-12: Automata marking the faulty and non-faulty sublanguages of L_1 after the backward Algorithm. The observation of $L_{1,f}$ is $\{o_1 o_2\}$ differs from the observation of $L_{1,nf}$ which is $\{o_1\}$

Chapter 4

Design, Simulation and Verification Tool

This chapter describes a design, simulation and verification software tool developed by the author during the research period. In order to show the motivation underlined the tool development a breve observation of existing publicly accessible tools is presented, as well as requirements for such software. Then an application of the tool is demonstrated on the example of industrial plant, which is described in the Chapter 1.

4.1 Requirements for Formal Tools

According to the survey, that was presented in [23], the industrial and research communities, *there is a need for light-weight formal methods and associated tools, where engineers can push-the-button and have powerful evaluations and analysis take place.* More then one decade later, this need has been even grown, and requirements the authors imposed are still actual. Shortly, the requirements are:

- Formal methods should be “invisible” and automatic
- Usage of open standards for sharing formal results
- Easy to use (either in academic or industrial field)

- Modular structure, lightweight architecture
- Scalable for real world problems

Nowadays, numerous commercial software tools can be found on market and even more accessible with no charge. Verilog hardware description language [18], Event-B and Roding [4], ProB model checker [17], Pessoa [11] to mention a few. These tools aimed for complex task and are capable to solve heave industrial-like problems. The lightweight academical response is GOAL [6], DESUMA [3], TCT [16], Supremica [14], etc.

Some of the tools are of high quality, other are easier to use. But the major drawback of the aforementioned software is that it is closed for modification, which makes it is almost impossible to use it for a research, for creation and validation of new formal techniques. An appropriate software tool for the studying and research purpose is desired to be easy to access, working on most of hardware and software platforms, flexible for changes, and with an intuitive user interface design.

Bearing in mind the above motivation, a specialised software tool was created in frame of this work. Its primary goal was a validation of concepts developed during the research. It provides simple access and zero-time enrollment time on general operational systems due to the exploiting web-technologies. It was thought as a Free Open Source Software (FOSS) tool with potential of enhancement by a wide community interested in it.

The main disadvantage of the web-based underlining approach is a relatively low performance due to the nature of the programming languages used. However, the last trends in web development show that this issue can be overcome in a near future.

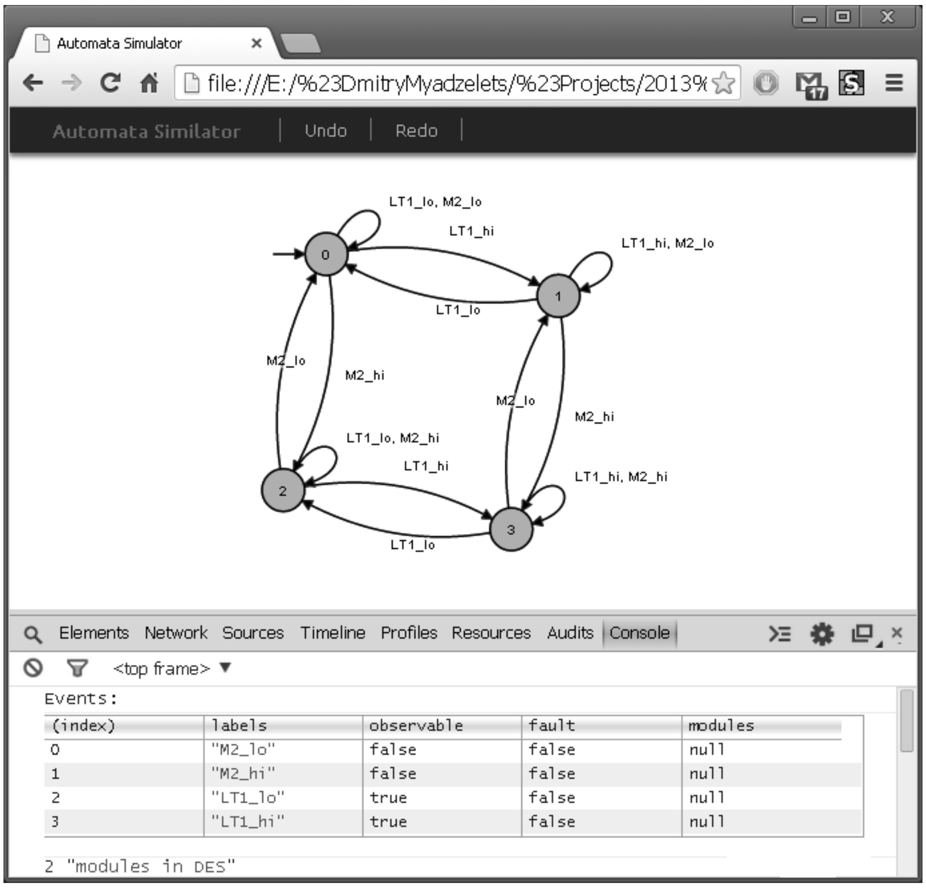
4.2 Tool Description

The charasterics of software tool developed during the research project are summarised in the Table 4.1. The objects and operations allow to cover many tasks while formal techniques development. Additional methods can be easily implemented.

As a programming language a general purpose web-oriented language was chosen, the JavaScript. For the sake of memory efficiency, a set–based approach was chosen for automata structural representation (see **??** for the approaches). The current graphical implementation exploits Scalable Vector Graphics (SVG) [15]. This allows use of any graphical representation (e.g. automata, influence diagrams) to be used with no translation in the most of types of documents. An instance of the screen of the tool is shown in the Figure 4-1.

Table 4.1: Objects and operations available with the tool

Object	Operations
Sets	Binary, Numbers, Strings and Objects operations
Sets	Operations on triples (for graph edges/transitions)
Graph	Manipulations with edges and nodes
Graph	Breadth-First Search (BFS) in a graph
Graph	Depth-First Search (DFS) in a graph
Graph	Automatic layout
Automata	Parallel composition of two automata
Automata	Intersection (full synchronisation)
Automata	Kleene closure
Automata	Subtraction
Automata	Emptiness verification
Automata	Copying operation
Automata	Projection to a given set of events
Automata	Complement operation
Automata	Reachability operation
DES	Centralized management of the events set
DES	Creation of a module in frame of the DES
DES	Creation of an external module
DES	Computation of common events of two automata
DES	Representation of the system with as the inference diagram
Diagnosability	Fault states and events support
Diagnosability	Computations of automata deterministic w.r.t. a failure history
Diagnosability	Computation of fault-reachable and fault-non reachable languages



Events:

(index)	labels	observable	fault	modules
0	"M2_lo"	false	false	null
1	"M2_hi"	false	false	null
2	"LT1_lo"	true	false	null
3	"LT1_hi"	true	false	null

```
2 "modules in DES"
```

Figure 4-1: View of the simulator in a browser window

4.3 Diagnosability verification of the system

In this section the problem of diagnosability verification of the system presented in the Chapter 1 is studied. In order to validate the theoretical approach and the algorithms introduced in this work a formal model of the system has to be constructed. The model contains automata of the component types listed in the Table 4.2. Moreover, the model contains automata which describe supervisory control policy for each control cycle.

Automata models of the system's physical components are constructed according to design patterns of the Generalized Device approach. Types of automata and composition techniques used to model this system are presented in Appendix A.

The resulting inference diagram of the system is depicted in the Figure 4-2. The diagram is built automatically by the simulation tool. It is interesting to see how

Table 4.2: Modules of the system depicted in the Figure 1-1

Module name	Number of modules
Butterfly valve	2
Gate valve	3
Screw conveyer	1
Vibrator	1
Belt conveyer	1
Fork level sensor	4
Air pump	1

the nodes which represent the control policy automata link all the nodes representing filed devices into the whole.

4.3.1 Global model of the system

Global monolithic model of a system is required for verification of centralized diagnosability. An attempt to construct the global model of a real complex system usually faces the states explosion problem.

The presented system with the current design consists of *53* automata. Using the simulation tool, an attempt to construct the monolithic model was made. Table 4.3 shows the growth of the automaton of the monolithic model by means of states, transitions and time spent for each iteration. The correspondent plot is depicted in the Figure 4-3.

The given system has relatively low complexity. Construction of the global model using another tool like NuSMV [10] is likely possible. However, the tool used in this case has performance not optimised yet, and the construction of the global model was performed only partially due to the time limit.

4.3.2 Example of a failure verification

For a demonstration of diagnosability verification with the developed tool the instance of a hypothetical technological failure is taken (though it could be a real case for the given system). A part of the material flow description, given by a technologist is

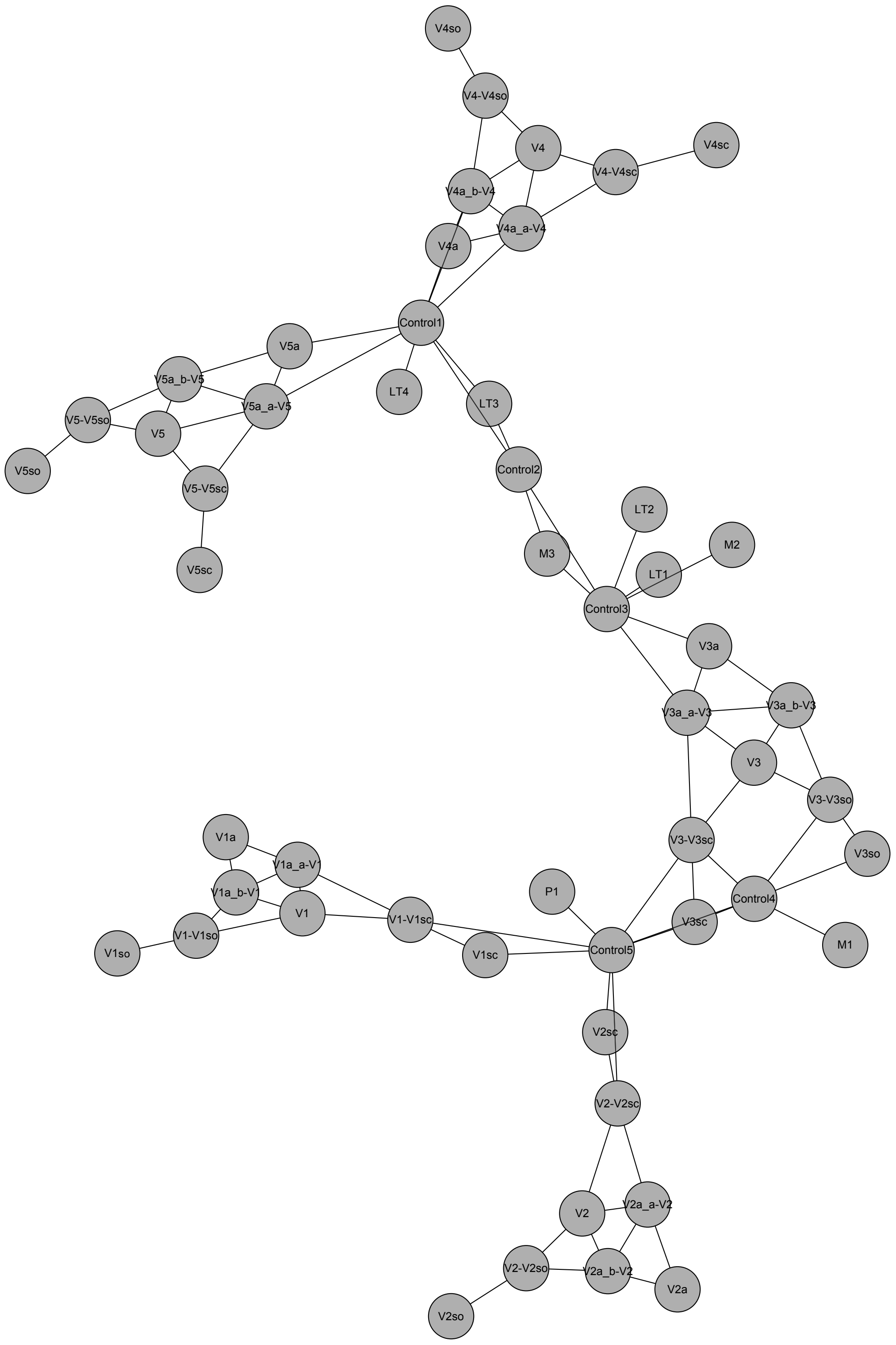


Figure 4-2: Inference diagram of the system depicted in the Figure 1-1

following.

Consider the P&I diagram of the system depicted in the Figure 1-1. Volumes of

Table 4.3: Growth of the centralized model during parallel composition

Number of modules	States	Transitions	Time spent, min:sec.ms
2	10	46	0:0.4
3	20	132	0:0.37
4	22	122	0:0.5
5	21	96	0:0.3
6	63	498	0:0.12
7	63	460	0:0.10
8	63	424	0:0.11
9	315	2939	0:0.78
10	630	7138	0:0.406
11	1260	16796	0:1.627
12	1386	17014	0:1.686
13	1323	14952	0:1.282
14	3969	58086	0:7.701
15	3969	55692	0:8.579
16	3969	53424	0:7.866
17	19845	318717	3:43.389
18	39690	716814	23:41.914

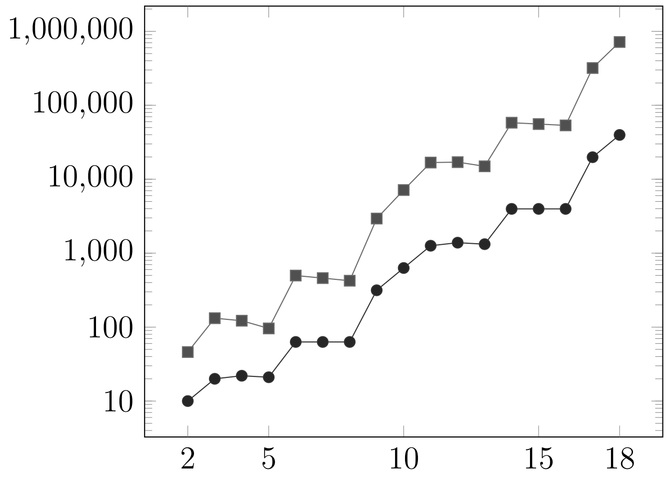


Figure 4-3: Growth of the number of states (curve with round dots) and transitions (curve with square dots) while the growth of the number of modules in the centralized systems model

the filter container and of the dust bucket are chosen according to a speed the filter container fills up, the dust bucket unloads and a speed of the material transportation by the both screw and belt conveyers. When the filter container is filled, it takes two times to load and unload the dust bucket. Due to physical limits of the process, the filter conveyer can not be emptied with one load–unload cycle of the dust bucket, neither the filter container can be full after two unload cycles. An automaton model which may describe this technological process is depicted in the Figure 4-4.

The automaton reflects also possible failures (shown with the marked state of the automaton). If the filter container is empty just after one unload, this may be a symptom of the screw conveyer breakage, destruction of the pipe or its connection to the valve. If the filter container is full after three unload cycles, it may be a sign of low dust bucket performance, pipes clogging, etc.

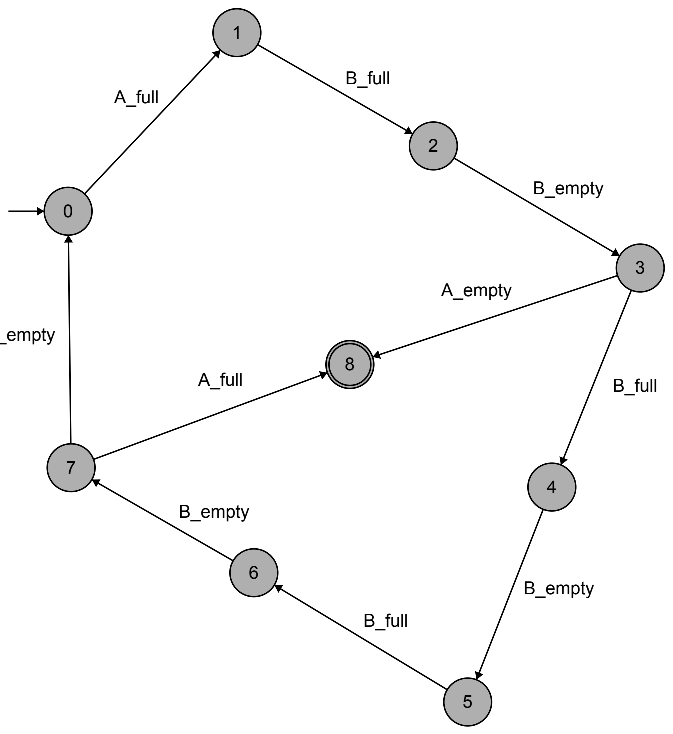


Figure 4-4: Model of a technological process failure

The failure automata model does not contain any observable events. Thus, the model is not diagnosable locally. However, the model can be linked to the fork level sensors $LT1$ and $LT3$ as with digital signals via cause-effect automata (see description in the Appendix A).

The diagnosability of the system was verified using the tool presented earlier. The forward-propagation algorithm performed 356 compositions of the modules in less then a second. During the failure propagation through the inference diagram (see Figure 4-2) six modules where updated while a little complexity growth was observed. The verification has shown that in the diagnosis of the failure two modules of the system participate, the sensors $LT1$ and $LT3$. According to the approach of this work, these modules can be treated as one virtual module. This virtual module can be used to build diagnoser for online monitoring of the system, particularly for detection of the above described failure.

Chapter 5

Conclusions and future work

"We are stuck with technology when what we really want is just stuff that works" – Douglas Adams, The Salmon of Doubt.

The main aim of this research was to bring the fascinating beauty of formal mathematical approaches closer to everyday practices of industrial development engineers in order to enhance the routine creation process of automated systems.

As shown in this work, the major obstacle for the application of formal methods in the industrial systems development process seems to be a lack of appropriate design techniques and tools. Despite the fact that the formal approaches are quite mature generally, and in the field of Discrete Event Systems particularly, probably the effort which is being taken in this area could be increased. As the confirmation, one can observe every-day design processes, decision assistance, programming tools and languages, which are often outdated from the perspective of what could be used nowadays, if such effort has had succeed.

To show the current situation in industry the Chapter 1 describes, with help of an industrial plant real example, what quantitative and qualitative methods are used nowadays besides the formal approaches in order to achieve desired properties of industrial systems.

Aa few goals were persuaded in this work. The first one is the modeling approach for design of manufacturing systems. The effort has been taken in development of formal representations of widely used industrial components. For the formal representation the framework of formal languages and automata has been chosen. The choice is based on a high availability of formal approaches which proved efficiency of this framework on the one hand, and a relative simplicity of the mathematical notation and graphical representation of automata models, on the other hand, bearing in mind that the results of the research should be understood by a wide community of industrial engineers with a little or no effort.

The automata design patterns, developed and validated during this research, incorporate both nominal and faulty behaviours. This approach appeared to be possible after decomposition of hardware components into small models, such that these models may efficiently abstract details of the formal model for the designer of an industrial system. In order to increase the level of abstraction, thus enhancing usability and reusability, the formal models are 'hidden' into UML-like blocks of Generalised Devices. This modeling process, as well as essential mathematical notation for discrete event systems in terms of languages and automata, and diagnosability types is described in Chapter 2.

Use of the formal methods is often corresponds with a problem of exponential explosion which may make the solution for the problem intractable. The diagnosability analysis is not an exception. Approaches which rely on modular nature of the complex system use a variety of notions and algorithmic techniques in order to tackle the computational burden. This work has introduced a new definition of diagnosability and notion of virtual modules. It is an algorithmic 'trick' to combine the existing modules of the system into new modules (which are not correspondent to the real components, and that is way are called “virtual”) in a way that the system with the new modularity is modular diagnosable. The theoretical part of this work is covered in Chapter 3.

The final goal of the research was to focus on applicability of the developed concepts. In order to validate the algorithms, which verify diagnosability and provide information for construction of virtual modules, a software tool has been created. It is described in the Chapter 4. There an application of this tool for an instance of failure in the real industrial process is demonstrated and compared with centralized approach.

Concluding, the developed modeling and new theoretical approaches have been validated with the software tool and proved efficiency of the concepts. Particularly, the performance advantage with comparison to the centralized approach is obvious. The advantages and disadvantages of the method with comparison to other algorithmic techniques, even if they differ from the theoretical point of view, has to be checked in the future. Additional effort is also required for performance optimisation of the developed software tool.

Appendix A

Automata models

Here a set of automata models for some industrial entities is given. The models are given both with no failure and failure behaviour. Failures are reflected as events according to the event-based failure modeling approach and as marked states for the state-based failure modeling approach. In the both cases a marked state in a figure means that a fault occurred.

A.1 Digital Inputs/Outputs, Sensors, Motors, etc., and their connection

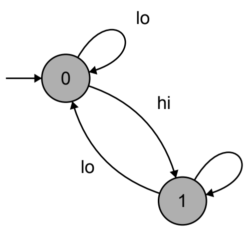


Figure A-1: General model of a digital input/output, relay, contactor, etc.

The Figure A-1 depicts a general model of a two-state component. This model reflects behaviour of digital inputs/outputs, relays, contactors, motors and other devices the behaviour of which can be equal to this abstraction. The Figure A-2 depicts a faulty version of the model. The faulty model reflects two failures: “stuck low”

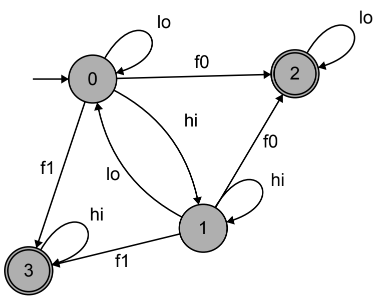


Figure A-2: Model of a digital input/output with failures

denoted as $f0$ and "stuck high" denoted as $f1$.

A consequent connection of two modules as the ones described above in a cause-effect manner imposes constrain on the "effect" module. Assume that there is two modules: 1 – "cause" (e.g. a contactor) and 2 – "effect" (e.g a motor). The model of the constrain and the result of composition of the entire system (two components and their constrain) is depicted in Figure A-3. If the module 1 has failures modeled as shown in Figure A-2, then the entire system looks as in Figure A-4.

A model of the same failure can be presented using the state-based failure approach. In this case no fault events are used. Since the model of a digital signal defines its full language, the model of the constrain has to be exploited instead, as shown in the Figure A-5. The resulting composition has lower complexity. It has a positive effect for the computation burden during verification of complex systems, since almost any industrial system includes many digital inputs, outputs and other similar components. However, it is arguable what model should include the failure of the component, the model of the component itself or models of constrains.

A consequent connection of two modules: 1 – "cause" (e.g. a contactor) and 2 – "effect" (e.g. a motor), where the second module has failures is depicted in Figure A-6. Here the failures are modeled using state-based failure approach.

A.2 Valve

The most known physical model of a valve is depicted in the Figure A-7. The model reflects a wide variety of valves, e.g. gate valves, butterfly valves, ball valves, etc.

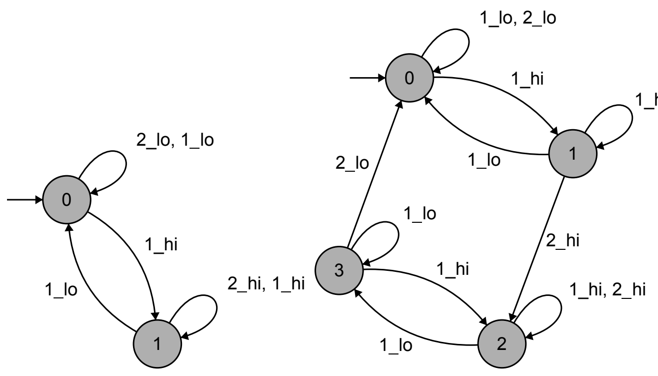


Figure A-3: The constrain model (left) for two consequent two-state components, and the result of the composition of the entire system (right)

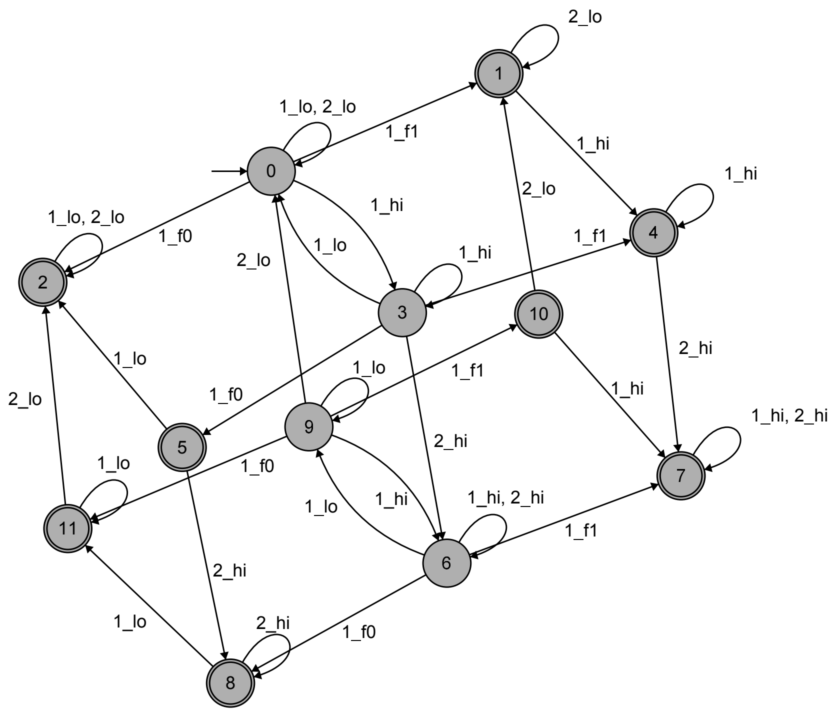


Figure A-4: Fault model of two consequent digital two-state modules. The first module has failures

In general, five states if this type of equipment can be distinguished: two boundary states (closed and open), two movement states (opening, closing) and stop in an intermediate position.

The “stop” position of the gate valve model can be marked as faulty, since gate valves must usually stay either closed or open.

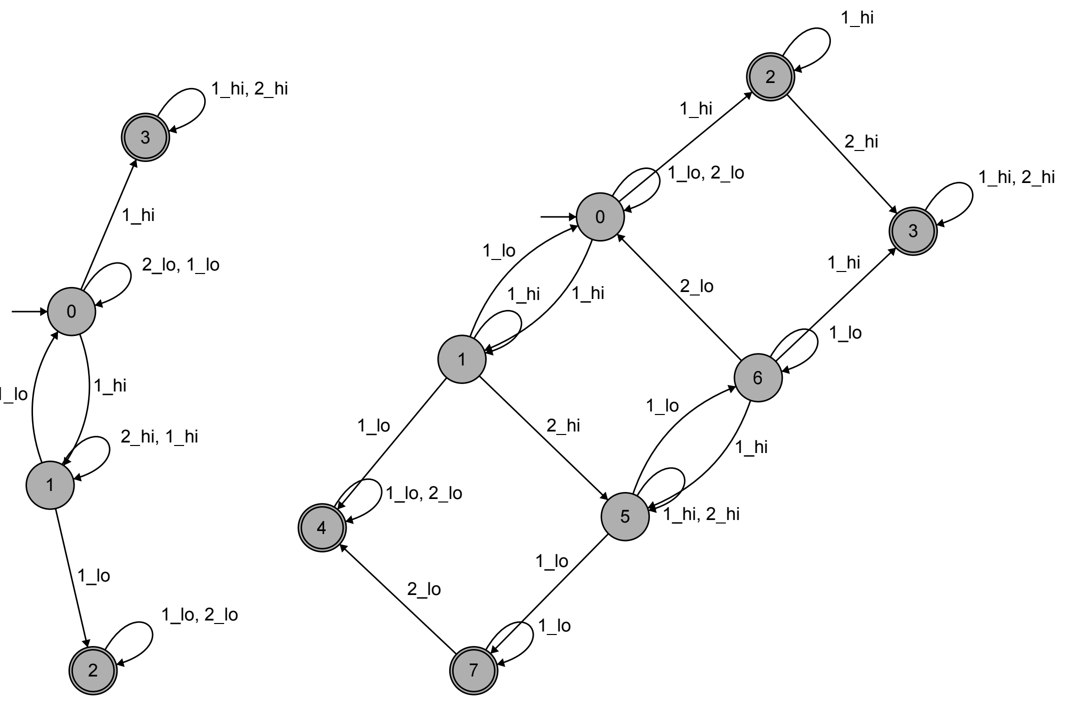


Figure A-5: The cause-effect fault automaton model (left), and the resulting composition of two components (right)

A.2.1 Valve with sensors

Figure A-8 shows the model of a sensor 1 built according to the model of two-state components described before, and a constrain model which assumes that the sensor observes “closed” state of the valve. From the perspective of the sensor the valve can be either closed or not. In another words, the valve is seen as a two-state component. Thus, the constrain automaton similar to the one shown in the Figure A-3 can be used.

Let the valve to have two sensor: one for the closed state and another one for the open state. An inference diagram of such system is shown in the Figure A-9. It reflects the coupling of all the correspondent models, e.g. valve, two sensors ($S1$, $S2$) and their constrains. The result of the composition of is depicted in the Figure A-10. The purpose of the figure is to show the level of complexity of such simple system.

A fault model of the valve's sensors is equal to the one depicted in the Figure A-2.

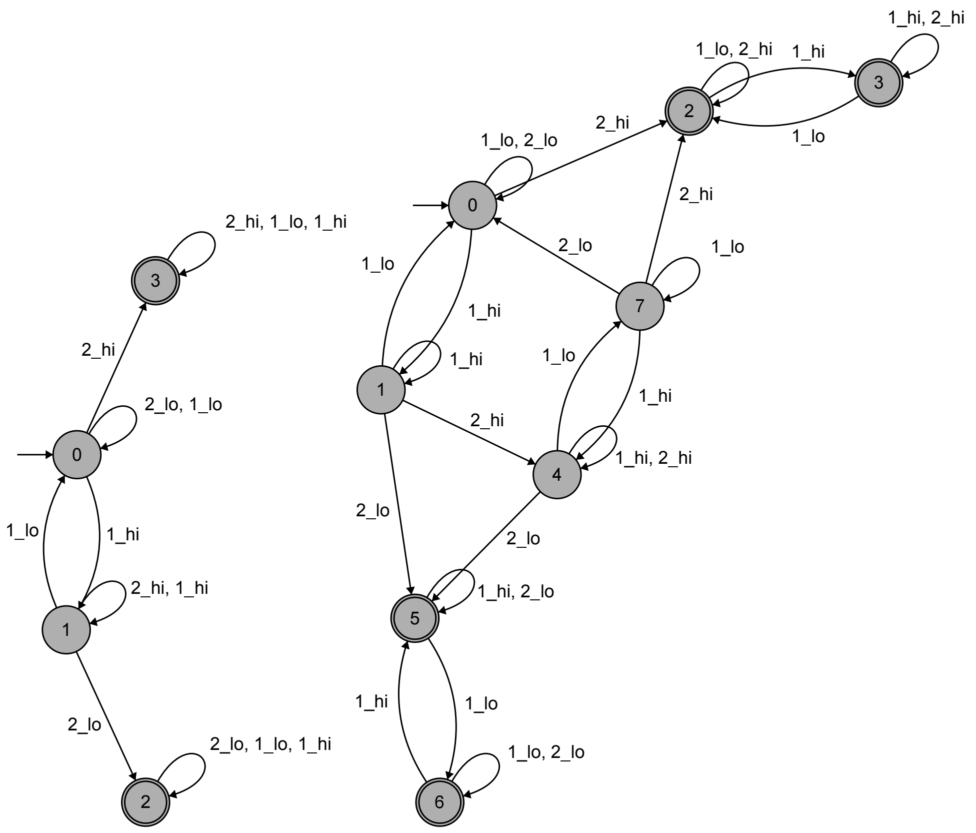


Figure A-6: The model of the constrain (left) and the composition result (right). Failures are in the second (i.e. "effect") module

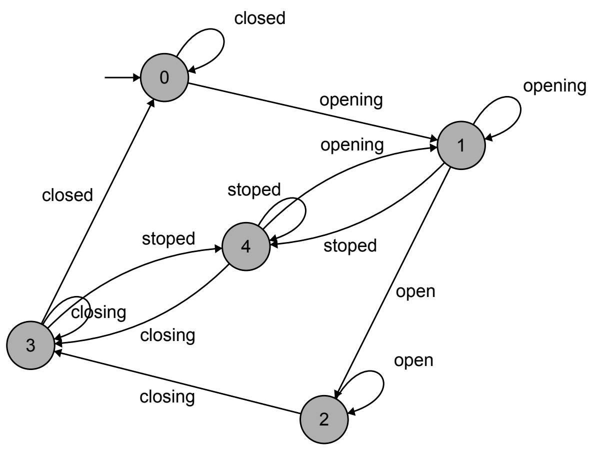


Figure A-7: Automaton model of a valve

A.2.2 Valve with actuator

Automaton model of a double actuation device for a valve is depicted in the Figure A-11 (think of an bidirectional motor). Actually, it has two actuating parts, denoted

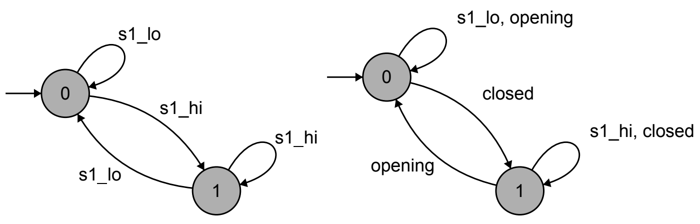


Figure A-8: Automata of a sensor and a constrain model for the sensor–valve relationship

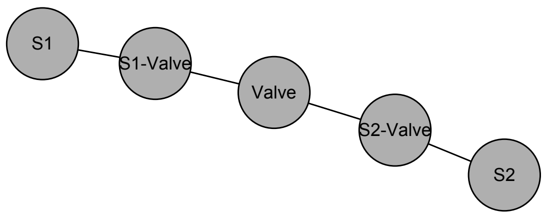


Figure A-9: Inference diagram of a valve with two sensors

in the figure by prefixes *a* and *b*. Part *a* is responsible for the "opening" movement, part *b* is responsible for the "closing" movement. Each actuating part is equal to the two-state digital component model, presented before. In the composition of these two components a consequent execution of events a_hi and b_hi is forbidden, i.e. it is necessary that $a\_hi \Rightarrow b\_lo$ and $b\_hi \Rightarrow a\_lo$.

The actuation devices are coupled with the physical valve model through a constrain automaton, similar to described above. The constrain automata for the valve are shown in the Figure A-12. The result of the composition of the valve, actuator and constrains is depicted in the Figure A-13. Marked states are correspond to the "stop" state of the valve, which may be considered as a fault for a gate valve.

A.2.3 Valve with sensors and actuator

Automaton which represents the complete model, i.e. the composition of the valve with two sensors and the actuator has 63 states and 424 transitions. A correspondent inference diagram is shown in the Figure A-14. Meaning of the nodes names is explained in the table below.

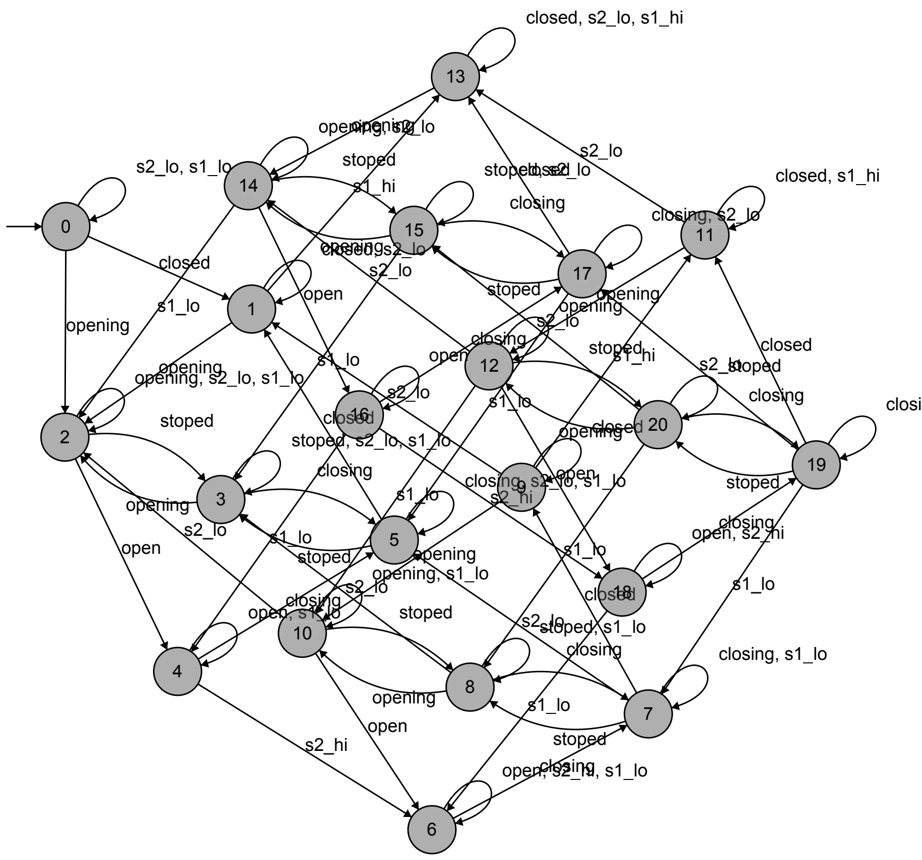


Figure A-10: Automaton of a valve with two sensors (open and closed)

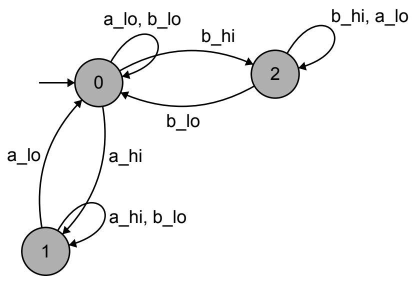


Figure A-11: Automaton of a valve actuator

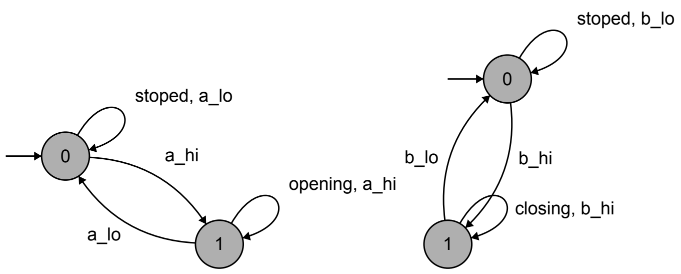


Figure A-12: Automata of the valve actuator constrains

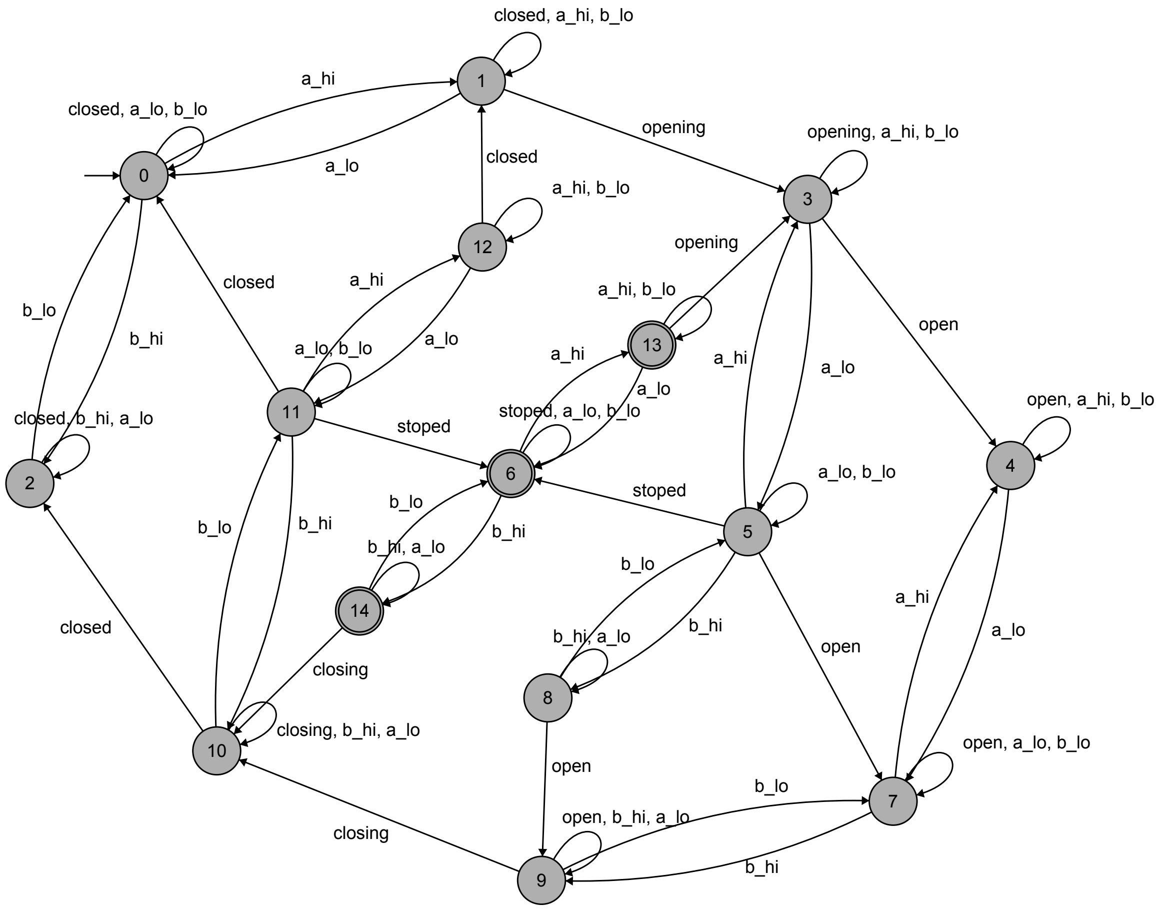


Figure A-13: Automaton of a valve with the actuator

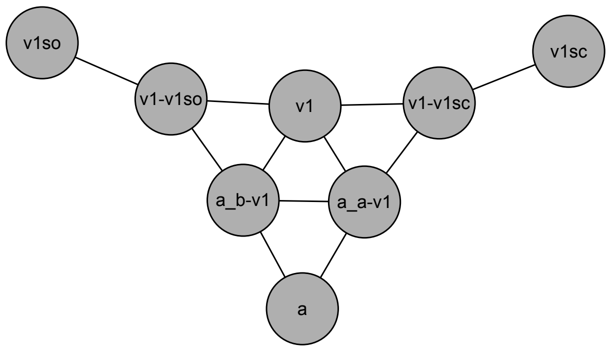


Figure A-14: Inference diagram of a valve with two sensor and an actuator (see Table A.1 for the nodes names description)

Table A.1: Nodes labels of the inference diagram of the valve, sensors and actuator

Node label	Description
v1	Valve
v1so	Sensor “Open”
v1sc	Sensor “Closed”
v1-v1so	Constrain of the valve to the sensor “Open”
v1-v1sc	Constrain of the valve to the sensor “Closed”
a	Actuator
a_a-v1	Constrain of the valve to the actuator’s part “Opening”
a_b-v1	Constrain of the valve to the actuator’s part “Closing”

Bibliography

[1] Atelier B - industrial tool for use of the B Method . http://www.atelierb.eu/.

[2] Center for Chemical Process Safety - Process Equipment Reliability Database (PERD). http://www.aiche.org/ccps/resources/perd.

[3] DESUMA Tool for discrete event systems. http://www.eecs.umich.edu/umdes/toolboxes.html.

[4] Event-B and the Rodin Platform. http://www.event-b.org/.

[5] Forensic Engineering and Failure Analysis - Case Studies A-Z List. http://www.intertek.com/forensics/casestudies/.

[6] GOAL is a graphical interactive tool for defining and manipulating Bchi automata and temporal logic formulae. http://goal.im.ntu.edu.tw/wiki/doku.php.

[7] International Electrotechnical Commission. Electropedia: The World's Online Electrotechnical Vocabulary. http://www.electropedia.org/.

[8] ISO/IEC 12207:2008 Standard for Information Technology - Software Life Cycle Processes. http://www.iso.org/iso/catalogue_detail?csnumber=43447.

[9] MATLAB, a high-level language and interactive environment for numerical computation, visualization, and programming. http://www.mathworks.it/.

[10] The NuSMV symbolic model checker. http://nusmv.fbk.eu/.

[11] Pessoa Software Toolbox for the Synthesis of Correct-by-Design Embedded Control Software. https://sites.google.com/a/cyphylab.ee.ucla.edu/pessoa/.

[12] SIEMENS Industry Online Support mean time between failures (mtbf) - list for simatic products.

[13] SIMATIC S7-300: the modular universal controller for the manufacturing industry. https://www.automation.siemens.com/mcms/programmable-logic-controller/en/simatic-s7-controller/s7-300/.

[14] Supremica Tool for Development Robust Control Systems. http://www.supremica.org/.

[15] SVG graphics format by the W3C SVG Working Group. http://www.w3.org/Graphics/SVG/.

[16] TCT design software for discrete event systems theory. http://www.control.toronto.edu/DES/.

[17] The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB/.

[18] Verilog hardware description language, IEEE 1364. http://www.verilog.com/.

[19] Alfred V. Aho and Jeffrey D. Ullman. The theory of languages. *Mathematical systems theory*, 2(2):97–125, June 1968.

[20] A. Arora, N.K. Medora, and J. Swart. Failures of Electrical/Electronic Components: Selected Case Studies. In *IEEE Symposium on Product Compliance Engineering, 2007. PSES 2007*, pages 1–6, October 2007.

[21] J. Banks. Introduction to simulation. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 7–13 vol.1, 1999.

[22] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, number 1536 in Lecture Notes in Computer Science, pages 81–102. Springer Berlin Heidelberg, January 1998.

[23] M.R. Blackburn and R.D. Busser. Requirements for industrial-strength formal method tools. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, 1998. Proceedings*, pages 137 –138, 1998.

[24] M. Bonfe and C. Fantuzzi. Design and verification of industrial logic controllers with UML and statecharts. In *Proceedings of 2003 IEEE Conference on Control Applications, 2003. CCA 2003*, volume 2, pages 1029 – 1034 vol.2, June 2003.

[25] Brualdi. *Introductory Combinatorics*. Pearson Education, 2004.

[26] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd ed. 2008 edition, October 2010.

[27] Anderson Chris. What is a Process Map? *Business Process Management*, 2014.

[28] E. M Clarke, Orna , Grumberg, and Doron Peled. *Model checking*. MIT Press, Cambridge, Mass., 1999.

[29] Olivier Contant, Stéphane Lafortune, and Demosthenis Teneketzis. Diagnosability of discrete event systems with modular structure. *Discrete Event Dynamic Systems*, 16(1):9–37, January 2006.

[30] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. In *1998 IEEE International Conference on Systems, Man, and Cybernetics, 1998*, volume 3, pages 3010–3011 vol.3, 1998.

[31] Wei Dong, Ji Wang, Xuan Qi, and Zhi-Chang Qi. Model checking UML statecharts. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 363 – 370, December 2001.

[32] Mian S. H. El-Tamimi A. M., Abidi M. H. and Aalam J. Analysis of performance measures of flexible manufacturing system. In *Journal of King Saud University-Engineering Sciences*, pages 115–=129 vol. 24, 2012.

[33] Eugenio Faldella, Andrea Paoli, Matteo Sartini, and Andrea Tilli. Hierarchical control architectures in industrial automation: a design approach based on the generalized actuator concept. Seoul, Korea, 2008.

[34] Joyce Farrell. *An object-oriented approach to programming logic and design.* Course Technology/Cengage Learning, Boston, MA, 2013.

[35] G. Frey and M.B. Younis. A re-engineering approach for PLC programs using finite automata and UML. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004*, pages 24 –29, November 2004.

[36] C. Michael Holloway. Why engineers should consider formal methods. In *In 1997 AIAA/IEEE 16th Digital Avionics Systems Conference*, page 9, 1997.

[37] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson/Addison Wesley, 2007.

[38] M Neil James. Failure Analysis - Industrial Case Studies. http://www.fatiguefracture.com/.

[39] Shengbing Jiang, Zhongdong Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 46(8):1318 –1321, August 2001.

[40] Meng Li and R. Kumar. Stateflow to extended finite automata translation. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 1 –6, July 2011.

[41] D. Maclay. Click and code [automatic code generation]. *IEE Review*, 46(3):25–28, May 2000.

[42] W.E. McUmber and B. H C Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering, 2001. ICSE 2001*, pages 433–442, May 2001.

[43] A. Otto and K. Hellmann. IEC 61131: A general overview and emerging trends. *IEEE Industrial Electronics Magazine*, 3(4):27–31, December 2009.

[44] Yannick Pencolé and Marie-Odile Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence*, 164(12):121–170, May 2005.

[45] Wenbin Qiu and R. Kumar. Decentralized failure diagnosis of discrete event systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 36(2):384 – 395, March 2006.

[46] Rajarshi Ray. *Automated translation of MATLAB Simulink/Stateflow models to an Intermediate format in HyVisual.* Chennai Mathematical Institute, Chennai, 2007.

[47] I. Ruiz, E. Paniagua, J. Alberto, and J. Sanabria. State analysis: an alternative approach to FMEA, FTA and markov analysis. In *Reliability and Maintainability Symposium, 2000. Proceedings. Annual*, pages 370–375, 2000.

[48] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995.

[49] M. Sartini, A. Paoli, R.C. Hill, and S. Lafortune. A methodology for modular model-building in discrete automation. In *2010 IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1 –8, September 2010.

[50] Matteo Sartini. *Architectures and design patterns for functional design of logic control and diagnostics in industrial automation.* Tesi di dottorato, Univercity of Bologna, March 2010.

[51] C. Secchi, M. Bonfe, and C. Fantuzzi. On the use of UML for modeling mechatronic systems. *IEEE Transactions on Automation Science and Engineering*, 4(1):105–113, January 2007.

[52] B. Sharda and S.J. Bury. Best practices for effective application of discrete event simulation in the process industries. In *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pages 2315–2324, December 2011.

[53] J. M Spivey. *The Z notation: a reference manual.* Prentice Hall, New York, 1992.

[54] R. Su, W.M. Wonham, J. Kurien, and X. Koutsoukos. Distributed diagnosis for qualitative systems. In *Sixth International Workshop on Discrete Event Systems, 2002. Proceedings*, pages 169–174, 2002.

[55] Rong Su and W.M. Wonham. Global and local consistencies in distributed fault diagnosis for discrete-event systems. *IEEE Transactions on Automatic Control*, 50(12):1923–1935, 2005.

[56] Nancy R. Tague. *The Quality Toolbox, Second Edition.* ASQ Quality Press, October 2010.

[57] V. Vyatkin. The IEC 61499 standard and its semantics. *IEEE Industrial Electronics Magazine*, 3(4):40–48, December 2009.

[58] V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, August 2013.

[59] L. Warrington, J.A. Jones, and N. Davis. Modelling of maintenance, within discrete event simulation. In *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pages 260–265, 2002.

[60] H.H. Weatherford and C.W. Brice. Simulation of industrial AC drive system under fault conditions. In *Eighteenth Annual IEEE Applied Power Electronics Conference and Exposition, 2003. APEC '03*, volume 1, pages 457–463 vol.1, February 2003.

[61] Tae-Sic Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially observed discrete-event systems. *IEEE Transactions on Automatic Control*, 47(9):1491 – 1495, September 2002.

[62] C. Zhou, R. Kumar, and R.S. Sreenivas. Decentralized modular diagnosis of concurrent discrete event systems. In *9th International Workshop on Discrete Event Systems, 2008. WODES 2008*, pages 388 –393, May 2008.

[63] Changyan Zhou and Ratnesh Kumar. Semantic translation of simulink diagrams to Input/Output extended finite automata. *Discrete Event Dynamic Systems*, 22(2):223–247, June 2012.