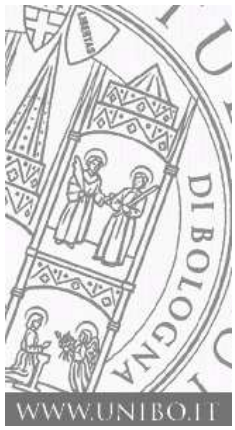
Architectures and design patterns for functional design of logic control and diagnostics in industrial automation

Ph.D. Thesis

Matteo Sartini

Tutor: *Prof. Claudio Bonivento*

Coordinator: *Prof. Claudio Melchiorri*



University of Bologna

XXII Ciclo

A.A. 2006 – 2009

Architectures and design patterns for functional design of logic control and diagnostics in industrial automation.

Ph.D. Thesis

Matteo Sartini

Tutor: *Prof. Claudio Bonivento*

Coordinator: *Prof. Claudio Melchiorri*

University of Bologna

XXII Ciclo

A.A. 2006 – 2009

Keywords:

Architectural Design Patterns, Fault Diagnosis, Discrete Event Systems, Automated Manufacturing Systems, Model Driven Engineering.

Ing. Matteo Sartini
CASY - DEIS - University of Bologna
Viale Pepoli 3/2, 40136 Bologna.
Phone: +39 051 2093870, Fax: +39 051 2093871
Email: `matteo.sartini@unibo.it`
URL: `http://www-lar.deis.unibo.it/people/msartini`

This thesis has been written in LaTeX.

Acknowledgments:
This work has been partially funded by the European Artemis Joint Undertaking funded project **CESAR:** *Cost-Efficient methods and processes for SAfety Relevant embedded systems*, sponsored by the European Commission in the IST programme 2008 of the 7th EC framework programme (ARTEMIS-2008-1).

This work has been partially funded by MIUR (Ministero dell'istruzione, dell'università e della ricerca).

To Anna

SOONER OR LATER,
THE WORST POSSIBLE COMBINATION OF CIRCUMSTANCES
WILL HAPPEN.
SODD'S LAW.

IN THEORY THERE IS NO DIFFERENCE BETWEEN THEORY AND PRACTICE.
IN PRACTICE THERE IS.
YOGI BERRA.

Contents

List of Figures

List of Tables

Preface

Recently in most of the industrial automation process an ever increasing degree of automation has been observed. This increasing is motivated by the higher requirement of systems with great performance in terms of quality of products/services generated, productivity, efficiency and low costs in the design, realization and maintenance. This trend in the growth of complex automation systems is rapidly spreading over automated manufacturing systems (AMS), where the integration of the mechanical and electronic technology, typical of the Mechatronics (see [87] and [8]), is merging with other technologies such as Informatics and the communication networks. An AMS is a very complex system that can be thought constituted by a set of flexible working stations, one or more transportation systems. To understand how this machine are important in our society let considerate that every day most of us use bottles of water or soda, buy product in box like food or cigarets and so on. Another important consideration from its complexity derive from the fact that the the consortium of machine producers has estimated around 350 types of manufacturing machine (see [22]). A large number of manufacturing machine industry are presented in Italy and notably packaging machine industry, in particular a great concentration of this kind of industry is located in Bologna area; for this reason the Bologna area is called "packaging valley".
Usually, the various parts of the AMS interact among them in a concurrent and asynchronous way, and coordinate the parts of the machine to obtain a desiderated overall behaviour is an hard task (see for some example [71] and [14]). Often, this is the case in large scale systems, organized in a modular and distributed manner. Even if the success of a modern AMS from a functional and behavioural point of view is still to attribute to the design choices operated in the definition of the mechanical structure and electrical electronic architecture, the system that governs the control of the plant is becoming crucial, because of the large number of duties associated to it. Apart from the activity inherent to the automation of the machine cycles, the supervisory system is called to perform other main functions such as: emulating the behaviour of traditional mechanical members thus allowing a drastic constructive simplification of the machine and a crucial functional flexibility; dynamically adapting the control strategies according to the different productive needs and to the different operational scenarios; obtaining a high quality of the final product through the verification of the correctness of the processing; addressing the operator devoted to the machine to promptly and carefully take the actions devoted to establish or restore the optimal operating conditions; managing in real time information on diagnostics, as a support of the maintenance operations of the machine. The kind of facilities that designers can directly find on the market, in terms of software component libraries provides in fact an adequate support as regard the implementation of either top-level or bottom-level functionalities, typically pertaining to the domains of user-friendly HMIs, closed-loop regulation and motion

control, fieldbus-based interconnection of remote smart devices. What is still lacking is a reference framework comprising a comprehensive set of highly reusable logic control components that, focussing on the cross-cutting functionalities characterizing the automation domain, may help the designers in the process of modelling and structuring their applications according to the specific needs. Historically, the design and verification process for complex automated industrial systems is performed in empirical way, without a clear distinction between functional and technological-implementation concepts and without a systematic method to organically deal with the complete system. Traditionally, in the field of analog and digital control design and verification through formal and simulation tools have been adopted since a long time ago, at least for multivariable and/or nonlinear controllers for complex time-driven dynamics as in the fields of vehicles, aircrafts, robots, electric drives and complex power electronics equipments. Moving to the field of logic control, typical for industrial manufacturing automation, the design and verification process is approached in a completely different way, usually very "unstructured". No clear distinction between functions and implementations, between functional architectures and technological architectures and platforms is considered. Probably this difference is due to the different "dynamical framework"of logic control with respect to analog/digital control. As a matter of facts, in logic control discrete-events dynamics replace time-driven dynamics; hence most of the formal and mathematical tools of analog/digital control cannot be directly migrated to logic control to enlighten the distinction between functions and implementations. In addition, in the common view of application technicians, logic control design is strictly connected to the adopted implementation technology (relays in the past, software nowadays), leading again to a deep confusion among functional view and technological view.

In Industrial automation software engineering, concepts as modularity, encapsulation, composability and reusability are strongly emphasized and profitably realized in the so-called object-oriented methodologies. Industrial automation is receiving lately this approach, as testified by some IEC standards IEC 611313, IEC 61499 (see [45], and [46]) which have been considered in commercial products only recently. On the other hand, in the scientific and technical literature many contributions have been already proposed to establish a suitable modelling framework for industrial automation (see [9], [94], [86] and [30]). During last years it was possible to note a considerable growth in the exploitation of innovative concepts and technologies from ICT world in industrial automation systems. For what concerns the logic control design, Model Based Design (MBD) is being imported in industrial automation from software engineering field. Another key-point in industrial automated systems is the growth of requirements in terms of availability, reliability and safety for technological systems. In other words, the control system should not only deal with the nominal behaviour, but should also deal with other important duties, such as diagnosis and faults isolations, recovery and safety management. Indeed, together with high performance, in complex systems fault occurrences increase. This is a consequence of the fact that, as it typically occurs in reliable mechatronic systems, in complex systems such as AMS, together with reliable mechanical elements, an increasing number of electronic devices are also present, that are more vulnerable by their own nature. The diagnosis problem and the faults isolation in a generic dynamical system consists in the design of an elaboration unit that, appropriately processing the inputs and outputs of the dynamical system, is also capable of detecting incipient faults on the plant devices, reconfiguring the control system so as to guarantee satisfactory performance. The designer should be able to formally verify the product, certifying that, in its final implementation, it will perform its required function guarantying the desired level of reliability and safety; the next step is that of preventing faults and eventually reconfiguring the control system so that faults are tolerated.

On this topic an important improvement to formal verification of logic control, fault diagnosis and fault tolerant control results derive from Discrete Event Systems theory (see [17]).
The aim of this work is to define a design pattern and a control architecture to help the designer of control logic in industrial automated systems. The work starts with a brief discussion on main characteristics and description of industrial automated systems on **Chapter 1**. In **Chapter 2** a survey on the state of the software engineering paradigm applied to industrial automation is discussed. **Chapter 3** presentes a architecture for industrial automated systems based on the new concept of *Generalized Actuator* (see [27], [72] and [90]) showing its benefits, while in **Chapter 4** this architecture is refined using a novel entity, the *Generalized Device* (see [28]) in order to have a better reusability and modularity of the control logic. In **Chapter 5** a new approach will be present based on Discrete Event Systems for the problem of software formal verification and an active fault tolerant control architecture using online diagnostic ([70]). Finally conclusive remarks and some ideas on new directions to explore are given.
In **Appendix A** are briefly reported some concepts and results about Discrete Event Systems which should help the reader in understanding some crucial points in chapter 5; while in **Appendix B** an overview on the experimental testbed of the Laboratory of Automation of University of Bologna, is reported to validated the approach presented in chapter 3, chapter 4 and chapter 5. In **Appendix C** some components model used in chapter 5 for formal verification are reported.

⋆ ⋆ ⋆⋆

First of all I would like to thank my supervisor, Prof. Claudio Bonivento, who lead me three years ago inside this major and who places its trust on me since the first day of this period. I can not exempt myself from thanking Andrea Paoli that following me inside all my three years with priceless teaching and suggestions. A special thanks goes to Prof. Eugenio Faldella for its precious teaching on industrial automation field and Andrea Tilli for its advice and suggestions during my work.
A special thank goes to Prof. Stéphane Lafortune of the University of Michigan, for the continuous encouragement during my staying in Ann Arbor and for its teachings on discrete events systems. I always remember with pleasure the stimulating and formative discussions with Rick Hill. Thanks to the other members of University, Luca Gentili and Alessandro Macchelli.
I have to thanks my family that always support mer, my mum with her sweetness and my dad that pass on me the quality to get what I want with work hard.
Last but always first in my heart and my mind a heartfelt thanks goes to my wonderful wife Anna that always support me, trusting in me and encouraging me during my work and especially during my life.

Bologna, 15th of March, 2010 *Matteo*

Architectures and design patterns for functional design of logic control and diagnostics in industrial automation

AMS	**Automated Manufacturing System**
CAD	**Computer Aided Design**
CAE	**Computer Aided Engineering**
CAM	**Computer Aided Manufacturing**
COTS	**Commercial Off-The-Shelf**
CNC	**Computer Numerical Control**
DES	**Discrete Events Systems**
ECC	**Execution Control Chart**
FBD	**Function Block Diagram**
FMS	**Flexible Manufacturing System**
FSM	**Finite State Machine**
GA	**Generalized Actuator**
GAMP	**Good Automated Manufacturing Practice**
GD	**Generalized Device**
GEMMA	**Le Guide d'Étude des Modes de Marches et d'Arrêts**
GSP	**General Supervision Policy**
IEC	**International Electrotechnical Commission**
IL	**Instruction List**
ISA	**International Society of Automation**
ISPE	**International Society of Pharmaceutical Engineering**
LD	**Ladder Diagram**
LTL	**Linear Temporal Logic**
CTL	**Computation Tree Logic**
OMG	**Object Management Group**
OOP	**Object Oriented Programming**
OOPSA	**Object-Oriented Programming, Systems, Languages & Applications**
PCA	**Physical Constraint Automaton**
PLC	**Programable Logic controller**
PMMI	**Packaging Machinery Manufacturers Institute**
SFC	**Sequential Functional Chart**
ST	**Structured Text**
SYSML	**Systems Modeling Language**
UML	**Unified Modeling Language**
V& V	**Verification and Validation**

Table 1: List of acronyms used in the thesis.

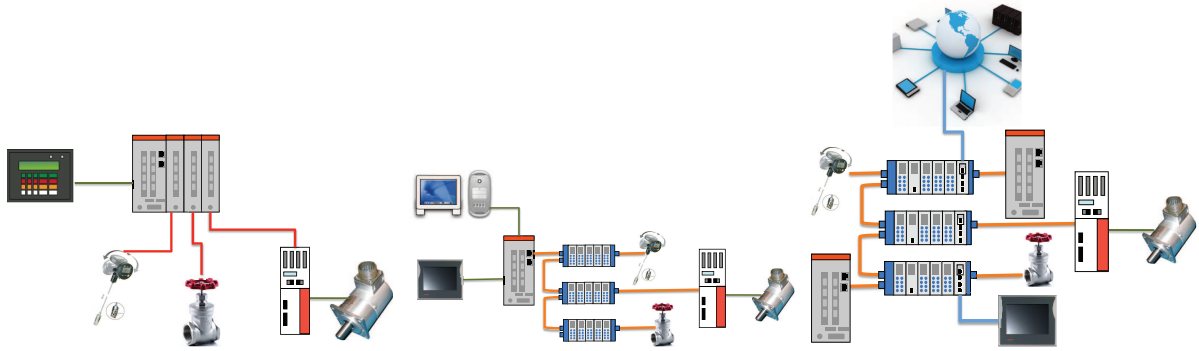
Chapter 1

Control design in industrial automated systems

This chapter showes the main characteristic of modern industrial automated systems. In last years there was a growth in the automation of production lines, implying the use of systems that are required to be ever more performing, reliable, flexible but at a lower price. Machines are then increasingly filled with embedded components constituted by engines, integrated in the mechanics, managed through a specific software command. The number and types of embedded components and technologies enabling specific functions (i.e. motion control, failure analysis, monitoring etc.) has raised the level of complexity of the whole systems with a clear impact on design process, that becomes a complex task, in which different technological fields interact (i.e. competences in mechanical engineering, in electronic engineering, in software and control engineering are needed to fulfil these tasks).

1.1 Design specification in industrial automated system

The enlarged demand of innovative and technologically advanced solutions in all industrial application domains has in recent years strongly promoted the development of powerful and versatile Automated Manufacturing Systems (AMSs), capable of working more reliably, with an increased product processing accuracy, and at a faster, sometimes astonishing, speed. The enhanced functionalities and overall performance characterizing the modern AMSs are, doubtless, primarily attributable to the novel ideas and effective solutions purposely conceived by the designers of their mechanical structure. In the achievement of these goals, however, also the design of the AMS control system plays a crucial role, because of the always more relevant and wider spectrum of activities delegated to it. Besides the automation of working cycles, other essential requirements that the control system is asked to deal with typically concern the need of:



(a) Centralized architecture. (b) Decentralized architecture. (c) Distributed architecture.

Figure 1.1: Control systems architecture evolution.

- emulate the behaviour of manifold mechanical devices (e.g., cams, gear wheels, etc.), so as to reduce the system structural complexity and limit downtimes due to automatic changeover of its constituent parts;
- dynamically adapt the control strategies according to different productive requirements and operational scenarios;
- ensure high quality of the finished product, making a thorough verification of its processing correctness while preventing useless workings and wastage of raw materials;
- support the operator with timely and precise indications about the actions that should be undertaken to establish or restore the desired operating conditions, ridding him of hazardous or burdensome jobs;
- provide a comprehensive set of real-time diagnostics information and pre-processed production data, adequately supporting system maintenance, raw materials and finished products handling, production organization and planning.

Such a broad set of functional requirements highlights and common experience makes it evident, that the design of the control system of a complex AMS is undoubtedly a hard task, involving a multidisciplinary cultural background [85]. Different considerations, however, should be pointed out as regards hardware and software design. From the former viewpoint, designers can profitably rely on technologically advanced commercial-off-the-shelf (COTS) resources, as powerful processing units, special-purpose controllers, smart I/O devices, networking infrastructures, which can be modularly composed to build up quite sophisticated and scalable architectures conforming to the application needs.

The hardware architectures of automated systems during last decades, like it's possible to see in figure 1.1, it's developed through different architectures. Centralized architectures (see fig. 1.1(a)), typically based on programmable logic controllers (PLCs) equipped with simple operator panels and supported in the accomplishment of complex or time-constrained tasks by special-purpose smart units (such as electrical axes controllers, simple HMI panale, ecc.), have dominated the hardware architecture up to the 80's. In the next decade, the spreading of decentralized architectures (see fig. 1.1(b)) is strongly promoted by the advent of the field-bus technology and by the use of industrial PC as a means to provide highly complex and expensive machines with definitely more user-friendly human-machine interfaces (HMIs). The ensuing

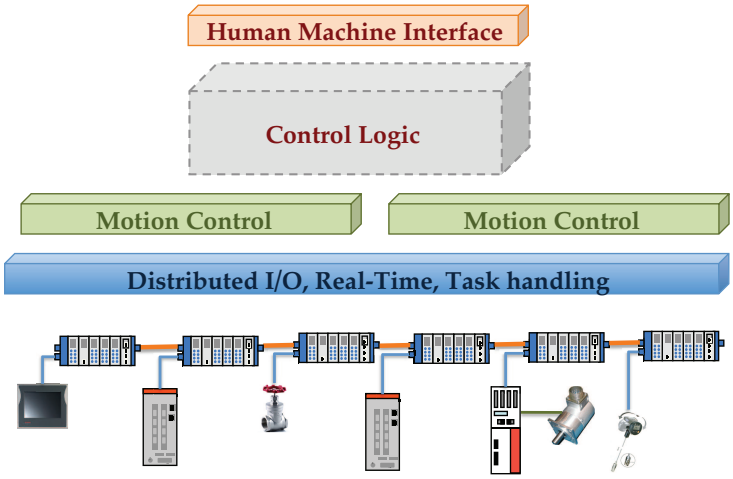
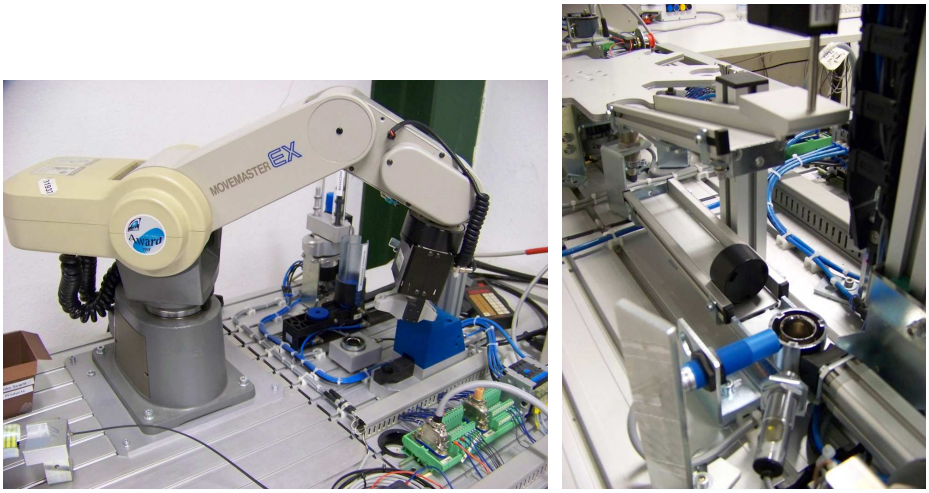


Figure 1.2: Control system architecture in automated systems.

availability of powerful multi-core PC-embedded systems, suitably enriched with soft-PLC environments and natively fitted out with the facilities needed for network interoperability, not only leads to the integration of both the control and HMI functionalities within a single platform, but also discloses the way towards fully distributed architectures (see fig. 1.1(c)). The overall automated system becomes not only a single machine but complex more different machine working together in a production lines.
Unfortunately, plain applicability of the "buy, plug & play" principle continues to have a narrower scope when the design of the control system from the software viewpoint is considered. The kind of facilities that designers can directly find on the market, in terms of COTS software component libraries, networking-oriented middleware, development tools and run-time environments, provide in fact an adequate support as regard the implementation of either top-level or bottom-level functionalities, typically pertaining to the domains of user-friendly HMIs, closed-loop regulation and motion control, fieldbus-based interconnection of remote smart devices. This functionalities are represented in figure 1.2, in a hierarchical representation where in the top level are presented the functionalities to interface the machine with the operator, and in the bottom level are presented the field functionalities. In the middle is presented the control logic of the automated system, for this part of control of the entire machine there are not availability and usually in this part is inserted the basic control of the machine during the nominal condition and all the diagnosis algorithms of fault detection. The "huge hole" existing in the middle needs to be properly filled in by software designers, still missing concrete support from vendors of industrial automation technologies as regards generally applicable frameworks, or at least application-level design patterns (in 2.2 wil be present a definition of design pattern), that can be profitably exploited (or assumed as reference guidelines) in the process of organizing and structuring the control logic of AMSs.

1.2 Classification of industrial automated systems

To better understand what are industrial automated systems and their complexity from a point of view of their mechanical architecture and control logic architecture it's possible to classify industrial automated systems based on its function and by the form of material handling sys-



(a) An example of assembly machine. (b) An example of testing machine.

Figure 1.3: Example of industrial automated machine.

tem. There are a wide number of functions that a machine can perform, but if one looks at the history of what has been built to date, and one was to try to classify the significant groupings that would result, several major classifications are:

- Assembly machines;
- Inspection machines;
- Test machines;
- Packaging machines;
- Computer numerical control (CNC) machines;
- Production processes.

This classification wants only try to defined different kind of automatic machine, of curse in complex production line we can found a mix of this machine.

1.2.1 Assembly machines

Assembly Machines as a group can range from the production of a high-volume part such as a spark plug or a piece of home kitchen cabinet hardware, to the construction of a cell phone. Throughput rates and product flexibility expectations can vary and usually changeover is required. Figure 1.3(a) shows a generic assembly machine, a robot that assembles short-stroke cylinders. There is the base part of the product brought into the workcell in some fashion. Parts feeders are used for the components to be added to the base part. And another method of transferring out the completed assembly is usually required. If the assembled part is a hinge for your home kitchen cabinets, then the output could be simply dumping them into a bin.

1.2.2 Inspection machines

Although in-process inspection is currently desired even more, inspection is often performed as an integral operation within the assembly machine. Computer vision systems and dimensional measurements are two of the commonly found inspections performed within the assembly machine. However, stand-alone inspection machines for checking a packaged product for the correct weight (check-weighers) and making sure no metal filing from all of the food processing machines fall into your box of corn flakes (metal detectors) do exist and have meaningful niche markets. Most of these inspection machines generically have a product inflow, a checking station, and two outflows. One of these outflows is the good product and the other outflow is for defective products. Depending on the product and its defect, the defective product may still be sold. If it is a food product and is only underweight, it can be eaten, possibly showing up in a factory seconds store. Obviously, products with metal filings are recycled, burned for heat value, or thrown away. Some examples of inspection process are:

- Checking one or more dimensions with mechanical gauging or electrical sensor;
- Checking one or more dimensions or features using a vision system;
- Checking weight for correct amount;
- Checking a liquid's volume by weight or level;
- Checking a filled Stand up Pouch (SUP) for leaks;
- Checking a product for metal filings, etc.;
- Checking free prize inside a box product.

The results from all of these can range from health risks (metal filings) to disappointed customers like a missing free prize. If we consider a production line to product wine bottle, of course the quality of wine bottle is depending from wine and bottling process, but from customer point of view and not perfect label alignment is a index of low quality of the wine bottle. An example of inspection machine is shown in figure 1.3(b), the machine test the colour of the workpiece and its height to decide if to works or not the workpiece.

1.2.3 Test machines

Machines that conduct some performance check on the filled, assembled, or processed product are sometimes referred to as test machines. Although some might argue that testing is part of inspection, the distinguishing feature is often the cycling of the product in some or all of its designed operation. In other words, an inspection machine functions by either a noncontact mode, or with a simple contact where some measurement or property is determined. A test machine makes the product do some action or work, such as cycling a spray head from a hand-powered misting bottle. The test is carried out on either a random basis, or on every spray head if trouble has been observed in the past, but the spray head is either passed as working properly, or is rejected. As opposed to inspection machines being potentially integrated into an assembly machine, most test machines are separate from the assembly process. Test machines are often highly specialized to the product being assembled or processed, and the devices used to perform the test and to judge the results cannot be easily integrated into the other machines. Figure 4.3 shows a test machine that is checking the previously assembled widget to see if it will hold together or whether it will fall apart. Following the example of testing the spray head,

there would need to be devices to move a single spray head into the test station in the correct orientation, an actuator to perform the test, a device to advance spray heads that pass the test, and another device to dump a rejected spray head into a hopper. The controller may need to be smart enough to allow the spray head to be actuated a variable number of times, so as not to reject heads that are good but not the best performers.

1.2.4 Packaging machines

Any finished consumer product of any value gets packaged in one of many different types of packages. It can be bags, boxes, cartons, aseptic boxes etc.. None of these packages significantly improves the performance of the product inside, but the packaging does help the consumer understand the product, differentiate the product from the competition, and improve sales dramatically. The area of packaging results quite large some typical packaging machines are machines devote to:

- Closing filled corrugated cardboard boxes;
- Filling bottles with liquids;
- Filling bags with dry products;
- Placing products into cartons;
- Weighing products for accuracy;
- Metal detection for safe consumption by consumer.

As it's to note the last two machines have been discussed in the previous section, this fact depend that it's not possible to have a complete division from machines and a complex production line is a composition of different kind of machine. To understand how is important packaging machine, we can think about how many product from packaging machine are in our daily life. Every day most of us use bottles filled with liquids, buy product in box like biscuits or cigarets, so there is a very large number and different kind of manufacturing machine.

1.2.5 Computer numerical control machines

Computer numerical control refers to the automation of machine tools that are operated by abstractly programmed commands encoded on a storage medium, as opposed to manually controlled via handwheels or levers, or mechanically automated via cams alone. The first CNC machines were built in the 1940s and '50s, based on existing tools that were modified with motors that moved the controls to follow points fed into the system on paper tape. These early servomechanisms were rapidly augmented with analog and digital computers, creating the modern computer numerical controlled machine tools that have revolutionized the design process. In modern CNC systems, end-to-end component design is highly automated using CAD/CAM programs. The programs produce a computer file that is interpreted to extract the commands needed to operate a particular machine, and then loaded into the CNC machines for production. Since any particular component might require the use of a number of different tools (as drills, saws, etc.) modern machines often combine multiple tools into a single "cell". In other cases, a number of different machines are used with an external controller and human or robotic operators that move the component from machine to machine. In either case, the complex series of steps needed to produce any part is highly automated and produces a

Figure 1.4: Example of a classic CNC machine.

part that closely matches the original CAD design. In figure 1.4 an example of a classic CNC machine.

1.2.6 Production Processes

Production processes are procedures involving chemical or mechanical steps to aid in the manufacture of an item or items, usually carried out on a very large scale. Most processes make the production of an otherwise rare material vastly cheaper in price, thus changing it into a commodity; i.e. the process makes it economically feasible for society to use the material on a large scales, in machinery, or a substantial amount of raw materials, in comparison to batch or craft processes. Production of a specific material may involve more than one type of process. Typical examples of production processes are chemical process, like in figure 1.5 where is shown an example of a gasification process, pharmaceutical process, industrial heavy etc. Production process can be roughly categorized as one of three types: batch, continuos, and discrete.

Batch Processes this kind of process are characterized by generation of finite quantities of material, called a batch, at each production cycle. Material is produced by subjecting specific quantities of input materials to a specified order of processing actions using one or more pieces of equipment. The batch goes through discrete and different steps as it is transformed from raw materials, to intermediates, and to final materials. Processed material is often moved, in total, between different vessels for different processing steps. Control of batch processes is not discrete or continuous, but it has the characteristics of both. Many pharmaceutical, specialty chemical, food, and consumer packaged goods are batch processes. They may be batch processes because the underlying chemistry or physics can only be done on all the material at once. Some batch processes are defined as batch

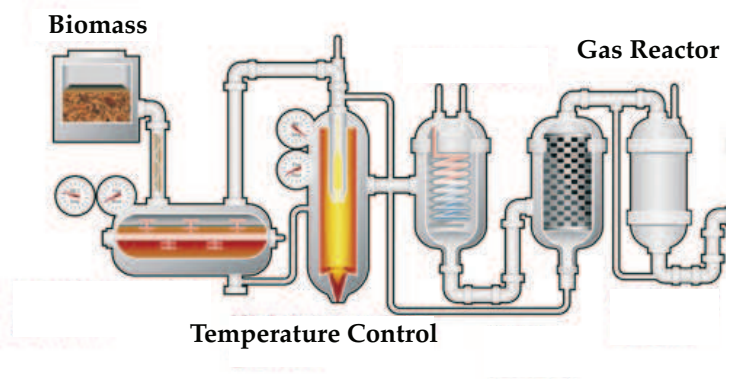


Figure 1.5: Example of production process machine for a gasification process.

because there are more product types than production lines, and each production line must be able to produce several different products. This is common in electronics, semiconductors, food processing, consumer products, and specialty chemical production. For example, the same production equipment may produce batches of chocolate cookies in the morning and sugar cookies in the afternoon.

Continuos Processes this kind of processes are characterized by the production of material in a continuous flow. Continuous processes deal with materials that are measured by weight or volume, without a discrete identity for any part of the produced material. Materials pass through different pieces of specialized equipment, each piece operating in a steady state and performing one dedicated part of the complete process. Once running in a steady state, the process is not dependent on the length of time it operates. Many bulk chemicals are produced in continuous production systems. Startup, transition, and shutdown do not usually contribute to achieving the desired processing. Material produced during these times often does not meet end product quality specifications and must be handled separately. Startup, transition, and shutdown, however, are important events that require specific procedures to be followed for safe and efficient operations.

Discrete Processes Discrete processes are characterized by production of a specific quantity of material, where each element of the material can be uniquely identified. Discrete processes deal with material that is counted, or could be counted. In discrete processes, a specified quantity of material (maybe just one element) moves as an entity between different pieces of processing equipment. The assembly of computer circuit boards is an example of a discrete process. Usually, multiple elements are processed using the same equipment configuration and raw materials. Startup, transition, and shutdown often produce the desired end product. Startup, transition, and shutdown are usually tightly controlled because they significantly impact the overall equipment efficiency. These events require specific procedures to be followed to place production equipment in the right state to make the desired product.

A widen discussion and description of production process can be found in [15].

1.3 Conclusions

In chapter was presented an introduction of industrial automated systems. In section 1.1 a brief description of most important specification and task of an automated systems are reported and it was presented some easy and typical example to understand what are automated systems. Starting from this typical example a functionality characterization was presented, an interesting reader can found more information in [22],[91] [15]. A particular consideration is has to focus on packaging machine, to understand its importance the consortium of machine producers, is called Packaging Machinery Manufacturers Institute (PMMI) has estimated around 350 types of packing machine (see [22] and www.packexpo.com). Also in Italy are presented a large number of packaging machine industry, in particular a great concentration of industry is located around Bologna area; for this reason the area around Bologna is called "packaging valley".
From this brief introduction is clear how automated systems and in particular manufacturing systems are very complex systems, of sure the "success" of an automated industrial system, in terms of functionalities and performances, are strongly relies on its physical structure: i.e. on the choices made during design phase in terms of the mechanical structure and electrical electronic architecture, but how discussed in 1.1 the control system plays a key role for the achievement of the targeted performance. The aim of this thesis is define an architectures for the modern industrial automated systems to help the designer to design the control system.

Chapter 2

State of the art of software engineering in industrial automated systems

This chapter deals with an overview of the control design in industrial automated systems and how the modern software engineering tecquiniques, like object orienting programming, are applied to modern industrial automated systems.

2.1 Software architecture in industrial automation

As explained in the chapter before an automated manufactory system is a machine that performs autonomously an industrial process, in which materials and energy are transformed to produce either consumer goods or input to other manufacturing systems. The task of control design for manufacturing systems represent a challenging and interesting problem, since the application domain presents heterogeneous and requires engineering efforts that include different kind of technological skills. In fact, the development process of a manufacturing machine control system is composed of several sub-tasks in the field of mechanical engineering, electric/electronic engineering, systems theory and also computer science, as the whole systems is composed by mechanical parts for product handling, heterogenous sensors and actuators for motion control and overall supervision, and several special on general purpose digital controllers that must be adequately programmed to perform efficiently the control algorithms. One of the most important control hardware in industrial automation filed are Programmable logic controller (PLC). PLC or programmable controller is a digital computer used for automation of electromechanical processes,such as control of machinery on factory assembly lines, amusement rides, or lighting fixtures. PLCs are used in many industries and machines. Unlike general-purpose computers, the PLC is designed for multiple inputs and output arrangements, extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. Programs to control machine operation are typically stored in battery-backed or non-volatile memory. A PLC is an example of a real time system since output results must be

produced in response to input conditions within a bounded time, otherwise unintended operation will result. The PLC was invented in response to the needs of the American automotive manufacturing industry. Programmable logic controllers were initially adopted by the automotive industry where software revision replaced the re-wiring of hard-wired control panels when production models changed. Before the PLC, control, sequencing, and safety interlock logic for manufacturing automobiles was accomplished using hundreds or thousands of relays, cam timers, and drum sequencers and dedicated closed-loop controllers. The process for updating such facilities for the yearly model change-over was very time consuming and expensive, as electricians needed to individually rewire each and every relay (a complete description of PLC can be find in [7]).

As it is common practice in every engineering context dealing with complex systems, software designers tackle the problem relying on the "divide et impera" approach. Drawing inspiration from the fundamental principles of decomposition and abstraction, they proceed to partition the whole AMS control logic into manageable simpler components, usually organized according to a hierarchical multi-layer architecture that directly mirrors, at least to a certain extent, the mechanical structure and the sensorial-actuation equipment of the AMS itself. Many factors, however, often hinder software designers from targeting the decomposition process towards the definition of an architecture comprising a wide variety of modular and reusable components, as much as possible platform-independent and directly applicable in other similar contexts. Among the common claimed (some arguably) factors causing software design to be more adherent to the code-and-fix approach rather than properly focussed in the problem domain, the following ones seem generally the most relevant. First of all, the ancillary role attributed to the software designers activity, too often addressed at the implementation of quickly-operative vehicles used for experimentally ascertaining the validity and effectiveness of the choices made by the AMS mechanical and electrical design team. Secondly, the burden of compelling, sometimes inappropriate, time-to-delivery commitments, which not only preclude any form of brain storming and exchange of significant experience among software designers, but often direct them to take approaches targeted at application-specific objectives only. Finally, the limited expressive power of (most of) programming languages currently available for PLC-based or soft-PLC PC-based platforms, somehow precluding plain applicability of well-established principles and methodologies proper of the object-oriented design paradigm.

It is therefore not entirely surprising that the costs associated with the software development lifecycle generally grow well beyond the budgeted expectations. In order to help solve these problems, many interesting proposals have been recently reported in the scientific and technical literature. Among them, some (e.g., mechatronic approaches) aim at improving the cost-effectiveness of the overall design process, favoring and stimulating concurrent engineering, co-design and co-simulation. Others suggest the use of state-of-the-art modelling languages (e.g., UML) and automatic code generation tools to enhance the software design, development and maintenance process. Several propose formal models (e.g. compositional theories, model-checking, model extraction) or techniques for verification and validation of component software, as a viable means to enforce reliability and reduce debugging efforts. Particularly important is also the contribution of renowned International Committees and Organizations, paving the way towards a standardization of the design methodologies (e.g. the IEC 61131/61499 norms, the Model Driven Architecture, MDA, proposal). The work here referred takes the cue from the ascertainment that an effective solution to the mentioned problems cannot derive solely from a fully synergistic cooperation between all design team's members, as well as from the use of powerful CAD-CAE tools and integrated development environments. This work try to define a generally-applicable design patterns focussing on cross-cutting control logic func-

tionalities and suitably abstracting from application-specific details, may support the process of modelling, structuring and implementing highly modular and reusable components.

2.2 Design patterns in industrial automation

The aim of this thesis is define a design pattern for industrial automated systems, so it is important to understand what design patterns are, and what they mean. A pattern can be variously described as:

- a practice or a customary way of operation or behavior.
- a model considered worthy of imitation.
- a blueprint intended as a guide for making something.

All of these descriptions apply to design patterns. A design pattern is a blueprint intended as a guide for use in design processes. Specifically in the field of software engineering and programming, a design pattern is a repeatable solution to a commonly occurring problem. A design pattern is not a design. Instead, a design pattern is a template for how to solve complex problems that applies to different, but related, situations. A design pattern can be transformed directly into code. Design patterns exist in many areas, from architecture and construction, to software design and development.

- In architecture, there are design patterns in houses. For example, colonial-style houses have the same first and second floor layout, varying in details and size, but not in overall structure.
- In civil engineering, suspension bridges follow design patterns, varying in scale and details, but not in overall structure.
- In software engineering, computer-human interfaces follow design patterns for windows and mouse actions, varying in detail for each application, but not in overall structure.
- Novels and plays follow design patterns, such as a typical mystery or romance novel plot.

Design patterns are used everywhere in modern society. They allow us to reuse the knowledge and experience of others. Design patterns mean that we do not have to solve every problem from first principles, but can instead rely on the experience of others who have come up with reusable solutions. Design patterns are usually developed as solutions to problems that can be reapplied to related problems. The term *design pattern* was first defined by an architect Christopher Alexander. His book (see [3]) related to urban planning and building architectures. He claimed that the architectural and engineering methods did not fulfil the real demands of the users of the buildings and urban environment. In its work, Alexander explained that there was some uniform way of building houses and towns that were comfortable and suit for the users' need. Alexander defines *the pattern language*, a common language that is shared by the architects to define the patterns that occur repeatedly during the design of buildings. This language should not be confused with programming languages. It is merely a common vocabulary and semantics for architects to speak about best-practice designs that solve a common problems. There are several one sentence definitions of patterns by various authors, one used by Alexander is:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way that you can use that solution a million times over, without ever doing it the same way twice
In 1987 Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming at OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) [5], they used the Alexander's idea in Smalltalk programming Tektronix. The same idea was applied later by Erich Gamma to study the reuse of Object-Oriented software (see [36]). In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Design patterns reside in the domain of modules and interconnections. At a higher level there are Architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system. Not all software patterns are design patterns. For instance, algorithms solve computational problems rather than software design problems.
In industrial automated systems, automation patterns have been applied mainly in software engineering problems of the automation systems. This is due to the origins of patterns in object-oriented software engineering. In general, in an object-oriented automation software the patterns proposed by E. Gamma ([36]) should be applicable. These patterns are generic in object-oriented programming, and the context of industrial automation does not make an exception. Automation systems are, however, a bit different from other information processing systems. The context of industrial automation, or the application domain, has a dynamic and changing nature of the system configuration, high data intensity, real-time constraints of measurements and control as well as the heterogeneity of the systems in a production plant. Somewhat similar environments could be found from telecommunication networks, and therefore the design patterns could be thought to be similar. An example to describe an automation pattern i using formal and informal diagrams, the formal notation uses UML notation. UML is the Unified Modelling Language (see [66], [67] and [77]). UML defines a rich set of diagrams and methods for describing complex systems and complex system solutions. In next sections we will see different design patterns for industrial automated systems.

2.2.1 A design pattern for control process - S88

A first example of design patterns for automated systems is the standard ANSI/ISA 88 (called S88). S88 is a standard addressing batch process control, it is a design philosophy for describing equipment, and procedures. It was approved by the ISA in 1995 and it was adopted by the IEC in 1997 as IEC 61512-1[1].
S88 provided for the first time a well thought-out approach to flexible manufacturing that was accepted by automation, control, and process engineers. It is a consistent set of standards and terminology for batch control and defines the physical model, procedures, and recipes. The standard sought to address the following problems: lack of a universal model for batch control, difficulty in communicating user requirement, integration among batch automation suppliers, difficulty in batch control configuration. The first step of a process-automation project is to define the requirements, usually they are defined by functional specification to try to design the

[1]The official standard is ANSI/ISA-88.01-1995. Batch control Part 1: Models and terminology. The international equivalent is IEC 61512-1.

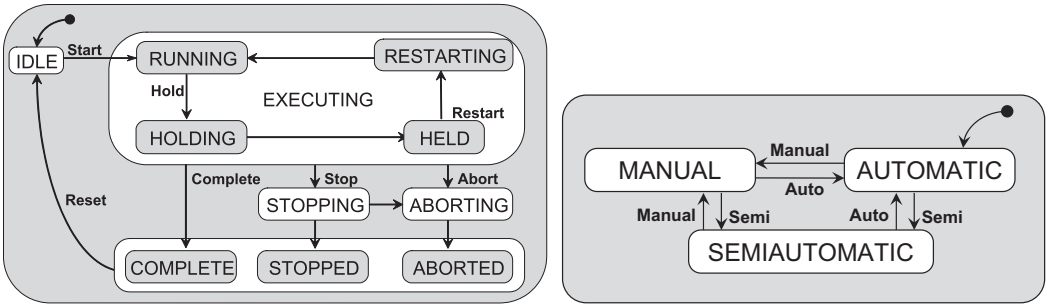
systems with a modular approach. A functional specification defines what the system should do, and what functions and facilities are to be provided. The GAMP (Good Automated Manufacturing Practice), a trademark of the International Society of Pharmaceutical Engineering (ISPE) describes a set of principles and procedures that help ensure that the product have the required quality. One of the core principles of GAMP is that quality cannot be tested into a batch of product but must be built into each stage of the manufacturing process (for a more exhaustive definition the reader is refereed to [37] [51]). For this reason it is important to have modular approach to design the entire systems (control logic, mechanical structure etc.).
The S88 standard try to define a sort of architecture separating the product definition information from production equipment capability. This separation wants define a procedure which allows same equipment can be use in different ways to make multiple products, or different equipment can be to be use to produce the same product. Product definition information is contained in recipes, and the production equipment capability is described using a hierarchical equipment model. This choice for the recipe/equipment separation is to try to make recipe development simple enough to be accomplished without requiring the services of a control systems engineer.
The recipe phase does not specify how the action is performed, the recipe only under what condition the phase is to be executed. S88 define 4 types of recipe but the most important are master and control recipes: master recipes are the templates used to create control recipes while control recipes are executed to produce a batch. The master recipe may specify information such as what types of equipment will be used, or what types of materials will be used. The control recipe has information added for the specific batch, such as what batch ID to assign to the batch, what material lot ID to assign to the produced material, and what equipment to use in production of the batch. The organization structure for equipment is called the equipment hierarchy. The S88 for the equipment hierarchy starts at the corporate level, called an enterprise in the standard and it arrives up to the production unit (to a complete definition see [49], and [50]). Equipment modules use equipment procedural control to achieve minor processing tasks. An equipment module is the container for performing the different elements of procedural logic required to achieve the process task. Equipment modules may be contained within units, or may be shared between units in a process cell. In either case, they usually contain all of the physical equipment and control capabilities to perform their minor processing function. The purpose of an equipment module is to coordinate and execute the procedural logic required to implement a phase, or to execute any other required equipment procedural control.
The S88 try to standardize how the different actions, the recipes, are executed defining the execution states. In the S88 the execution state is called *procedure's state*. The standard don not define all the states but it provides an example set of states but not establish a formal standard, an example of procedural control state is depicted in figure 2.1(a). The states usually have this signification:

Idle: The procedure element is available for execution.

Running: The procedure element has received a start command and is running its procedural logic.

Complete: The procedural element has completed normally.

Holding: The procedural element has received a hold command and is in the process of going to a stable held state. The procedural element may transition directly to held state, if no special actions are required.



(a) State model procedural control. (b) State model element modes.

Figure 2.1: Example of S88 design pattern.

Held: The procedural element is in a state suitable for a longterm delay that may be resumed later.

Restarting: The procedural element is performing any restart logic required to go from a held state to a running state. The procedural element may transition directly to running state, if no special actions are required, then .

Stopping: The procedural element has received a stop command and is transitioning to a stopped state as a controlled normal stop. The procedural element may transition directly to stopped state, if no special actions are required.

Stopped: The procedural element is no longer running and has performed a controlled normal stop.

Aborting: The procedural has received an abnormal stop (abort command) and is executing any aborting logic. The procedural element may transition directly to aborted state, if no special actions are required.

Aborted: The procedural element is no longer running and has performed an abnormal stop.

The states of machine of figure 2.1(a) evolves under the commands:

Start : Starts the procedural element. Used by an operator to start a control recipe, the logic in the procedure takes care of starting the lower-level elements.

Hold: Commands the procedural logic to go to a held state. Usually used by an operator to pause operation of the procedure logic.

Restart: Commands the procedure logic to release the hold and return to the running state.

Stop: Commands the procedure logic to stop executing and perform a normal shutdown.

Abort: Commands the procedure logic to stop executing and perform an abnormal shutdown.

Reset: Commands the procedural logic to reset and be ready to execute again. This command is usually performed by the recipe-execution system and is automatically sent after the system determines that the procedure element has completed.

S88 try to standardize also a "mode of operation". The mode determines how procedural elements respond to commands and how procedural control progresses. The modes can be represented in a state model as depicted in figure 2.1(b), this is an independent state model from the procedural element states. The mode defines what a recipe-execution system does with transitions between procedure steps. In automatic mode, the procedure logic is automatically executed. In manual mode, the operator determines what logic to perform. In semiautomatic mode, the operator decides when to step the logic.
While the ISA 88 standard has been applied to many non batch problems, the is no consistently defined methods for applying it in non stop-production. Non stop production is either continuos or discrete manufacturing where there are no breaks allowed in the product flow. Discrete production examples include the movement of discrete products , such as bottles, electronic components etc. Continuos production occurs when the product moves in a a flow there are no discrete countable elements a typical example are chemical production. To extend the S88 of this kind of process, a set of rules was defined to apply S88. This set of rules is called NS88, the goal of this approach is to maintain the concepts of separation of product definition, the recipes from intrinsic equipment capabilities. For a complete definition of this approach the reader is referred to [15].

2.2.2 A design pattern for manufacturing systems - GEMMA

Another example of design pattern is GEMMA (Le Guide d'Étude des Modes de Marches et d'Arrêts.), it means literally design guide for start and stop modes. GEMMA is a graphical checklist which allows the designer to define from the beginning all the operations and their consequences for a machine. The goal of Gemma is help the designer to define all the possible machine situation during its working and managing all the fault and emergency situation. GEMMA is a graphical tool and it borns as an helper to define a SFC project with a well-define structure. GEMMA is baed on three basic concepts: (i) Start modes are defined with the control part fully energized, (ii) The definition of production state (iii) Three general types of start and stop modes.
For GEMMA, the definition of start and stop modes affects the whole production and control parts of the machine, but from a control point of view, with the control part fully energized and i n working order. Basically, we can define two general modes: control part not energized and control part energized. The system, comprised of the machine and its control, will be in only one of those two states and can switch from one state t o the other through transitions. *Start modes* are defined with the control part fully energized. Any machine can be broken down into a productive part and a control part. Materials and energy are supplied to an automation production system, which produces transformed materials in accordance with the operator's orders and the environment. *The production state* is referred to a set state where the machine produces some work which adds value to materials which are fed into it; this is the added value. However, a machine is not 100% of the time in the production mode; it could be, for example, under repair, adjustment or modification. *Three general types of start and stop modes* are defined in GEMMA, in these three "ways" are groped all the different modes of the machine have been defined as:

Production Procedures: These include all necessary modes used too obtain the added value expected for the machine; they are not necessarily all productive modes (preparation mode) but are indispensable for production (before, during or after production).

Stop Procedures: A machine i s never operated 100% of its full useful life, so it is stopped it f

o r external reasons (end of the work period, lack of supply).

Failure Procedures: Any machine or system fails at one moment or another. These circumstances are described in the failure procedures, which cover all the internal reasons for stopping the machine. NOTE: The description of the states includes actions or states of any part of the machine, including the control part; it can also include special action to be taken by the operator or maintenance people themselves, such as manual action on the mechanisms, or writing in a log book, or reference to external written procedures, among others.

In figure 2.2 is depicted the model architecture of GEMMA where are emphasize the three types of start and stop, in the following a brief description of the three states groups:

Production Procedures

F1 Normal Production: This is the normal mode of production of the machine which, in this state, produces the transformed materials as the main and expected output of the machine.

F2 Start up: This state i s used for the machines which request special action, such as preheating, pressurization, prefilling or other, prior to production.

F3 Shut down: Some machines need to be emptied and cleaned before stopping or between production cycles.

F4 Unsequenced Test Mode: this mode allows the operation of some parts of the machine to be checked without following the usual sequence of operation. More commonly, this state can be called "human" control.

F5 Sequenced test mode: In this mode, the machine's cycle of operation can be followed step by step in the normal sequence. The machine can or cannot produce during this state.

F6 CALIBRATION MODE: This mode allows the instruments installed on the machine to be adjusted, set, calibrated or recalibrated.

Stop Procedures

A1 Initial State Stop: This corresponds to the zero energy state, with the machine de-energized but with the control circuits energized. Exist a clear relationship with GRAFCET initial state stop.

A2 Requested Stop at the End of Cycle: In this mode, the machine has been asked to stop at the end of the normal production cycle, i.e. to go into the A1 state, but production must be completed before then. A2 is a transient mode to A1.

A3 Requested stop in a determined state: The machine has been asked to continue production before stopping at a state other than the end of the production cycle. This a trancient state before A4.

Obtained stop: The machine i s stopped in a step other than the normal cycle's final position of the normal cycle.

A5 Preparation to Restart after a Failure: In this mode, all the requested actions (material, clearing, cleaning, change of broken tool, etc.) after a failure are effected before restarting.

A6 Production Reset to the Initial State: In this mode, the machine or its mechanism is set back, manually or automatically, to a position ready for resumption of production from the initial state.

Failure Procedures

D1 Emergency Stop: This mode regroups all the special sequences or actions which have

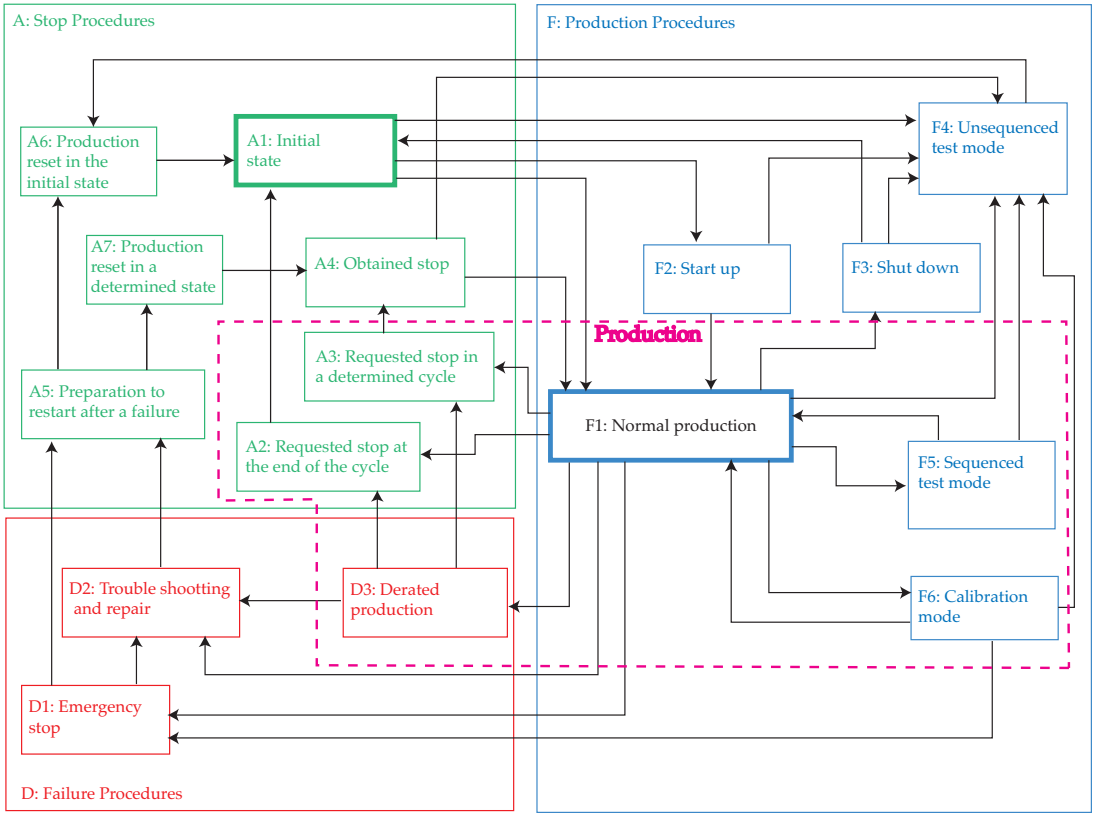


Figure 2.2: Gemma architecture state.

to be taken in any emergency conditions. This mode includes stops, but also special movements to limit the consequences of the emergency or the failure conditions.
D2 Trouble-shooting and Repair: In this mode the machine can be examined after the failure and actions taken to allow a restart.
D3 Derated Production: Under certain circumstances, it is necessary to continue production, even after a failure of parts of a machine. In this case, the production can be derated or forced; it could be effected by manual interventions of the operators, instead of automatically.

GEMMA is comprised of a set of "boxes" representing the different modes families F, A or D of the machine and showing the transitions between the boxes. Each box has a general name, chosen from the 16 general modes, and contains a specific function or action adapted to the machine itself. Probably the most important feature of GEMMA is help the designer in a structured design approach. This makes it suitable for process automation when one wants to take into account all the different factors and operation modes that may appear. Roughly speaking we can identify the following three modules when dealing with the design of an automated system: (i) Security module, (ii) Operation Modes module and (iii) Production module
The representation used within GEMMA takes into account these modules as well as the internal relations among them. The hierarchy shown in figure 2.2 tries to stress the security aspects of automated production systems. This will be the case in an emergency situation, device failure, or even when the production system; for any reason; is not generating the product properly. Under these situations the Security module has higher priority over the other mod-

ules in order to take the appropriate decision. It is also worth to notice the introduction of the operator as an integral part of the system. The operator adds experience in the switch from automatic to manual operation modes. This way the global control of the process can be the result of intermittent actuation within the *Manual mode* and the *Automated mode*. The Production module appears hierarchically under the previous ones. This module is the responsible for the sequential operation of the production process and operate son the basis of the state of the other modules. As it has been noticed before, GEMMA has a close connection with GRAFCET. Within this modular approach this means we will have different partial GRAFCET that will be needed to interconnect. Therefore, within the Production module we will start with the basic GRAFCET that implements the automation of the production system. This will be done without taking into consideration any other specification; just a sequential operation that implements a production cycle. This GRAFCET is known as the Production GRAFCET and will be the starting point for the application of GEMMA. On the other hand, within the Operation Modes module the operator takes decisions over the Production GRAFCET and can decide to enter into the control loop (by changing the operation mode to manual; or to leave the loop if the problem is solved by turning back to Automatic mode). This module will also have an associated GARFCET that is usually referred as the Conduction GRAFCET. Finally the Security module emergency situations and failures are taken into account. The protocol for an emergency stop is implemented as well as the convenient steps towards the take of the system to the initial conditions in order to restart the production. In this later case the associated GRAFCET is called the Security GRAFCET. As it is seen, one of the important parts of this structured approach is the presence of the operator and the possibility of introducing decisions over specific devices inside the *Security module* and the *Operation models* module. To a complete definition and description of GEMMA the interesting reader is refereed to [65] [19]

2.3 Standard language in industrial automation

Systematic approaches and development methodologies deriving from Software Engineering field have improved the programming practice in many application domains. However, large part of the success of analysis and design methods, especially those relying on the Object Orienting approach (OO approach), derives from the features of programming languages and environments supporting the same principles that drive the specification phase: modularity, encapsulation and information hiding. In fact, even if design models based on object-orientating may be implemented on non-OO languages, with some programming rules, constraints arid a certain amount of efforts, the results that cart be achieved with regard to maintenability and reusability depend considerably on the skill and programming experience of software developers. In practice, an efficient software development methodology should be supported by an implementation phase in which the design model is accomplished with minimal efforts within the constructs of the programming language. The standardization and portability features of that language, will furthermore help in the readability of code and module libraries within several different projects.
The main reasons that justify the slower improvements of PLC programming languages can be find in these points:

- Compatibility with old programming language is generally a desirable point, but this it is a “weight ” in term of evolution PLC programming.

- Software implementation is an important point, but industrial control are strictly con-

nected with I/O boards and are supposed to work in hard physical environment, so reliability and robustness of acquisition device are priority than software aspects.

- The hard connection between the PLC and the controlled physical world leads the programmer to adopt of low level mechanisms directly handle I/O and memory information, expecting that this improve computational performance.

Hardware vendors are in general affected by similar constraints with old hardware compatibility the result is that the PLC market is almost exclusively composed of "stand alone " solutions, with incompatible and proprietary tools. However, this historical scenario have had recently a turn towards standardization, thanks to the publication in 1993 of a document from the International Electrotechnical Committee (IEC 61131), which defines a standard set of terms, requirements, guidelines, programming languages, software and communication features specifically oriented to the domain of industrial controllers and PLC. This document, called IEC 61131 is divided into eight parts. The most well-known part of the IEC 61131 document, the third one, focus on the programming languages and is currently an International Standard, called IEC 61131-3, on which a large part of PLC vendors and users are more and more converging, in order to improve both the market offer and the features of programming tools for industrial controllers.
The importance of IEC 61131-3 and its contents is not only related to its attempt to unify, at least from a syntactical point of view, programming languages designed for PLC applications, but especially to the introduction of modularization and encapsulation structures that can improve the applicability of Software Engineering methods even in industrial domains, with particular regard to those driven by the Object-oriented approach. In last years a new standard has been presented in industrial automation world from International Electrotechnical Committee, the IEC 61499 a new standard to extend the IEC 61131 to distributed control systems. In next sections will show a briefly discussion on IEC 61131 standard and IEC 61499.

2.3.1 Standard language: IEC 61131-3

The IEC 61131-3 standard collects concepts and features derived from a wide set of preceding documents and standard widely adopted in the manufacturing industry, i.e. those related to the definition of GRAFCET (see [44]), but also from the consolidated practice of industrial control programming. The IEC 61131-3 defines a set of five different programming language and some common software feature as data type, software organization, execution control, etc. This standard represents a great progress in the domain of PLC and industrial control programming and the concepts and features that it prescribes cannot be disregarded, considering the current status of diffusion of the standard, in the definition of any design methodology for industrial control. The software model defined by the IEC 61131-3 standard covers all the aspects related to the modularization of control application, the definition of variables and data flow connections between program modules, computanional resources and with external I/O. The basic high-level language elements and their interrelationships are illustrated in figure 2.3. These consist of elements which are programmed using the languages defined in this standard, that is, programs and function blocks; and configuration elements, namely, configurations, resources, tasks, global variables, access paths, and instance-specific initializations, which support the installation of programmable controller programs into programmable controller systems. A configuration is the language element which corresponds to a programmable controller system as defined in IEC 61131-1. A resource corresponds to a "signal processing function" and its "man machine interface" and "sensor and actuator interface" functions (if any) as defined in

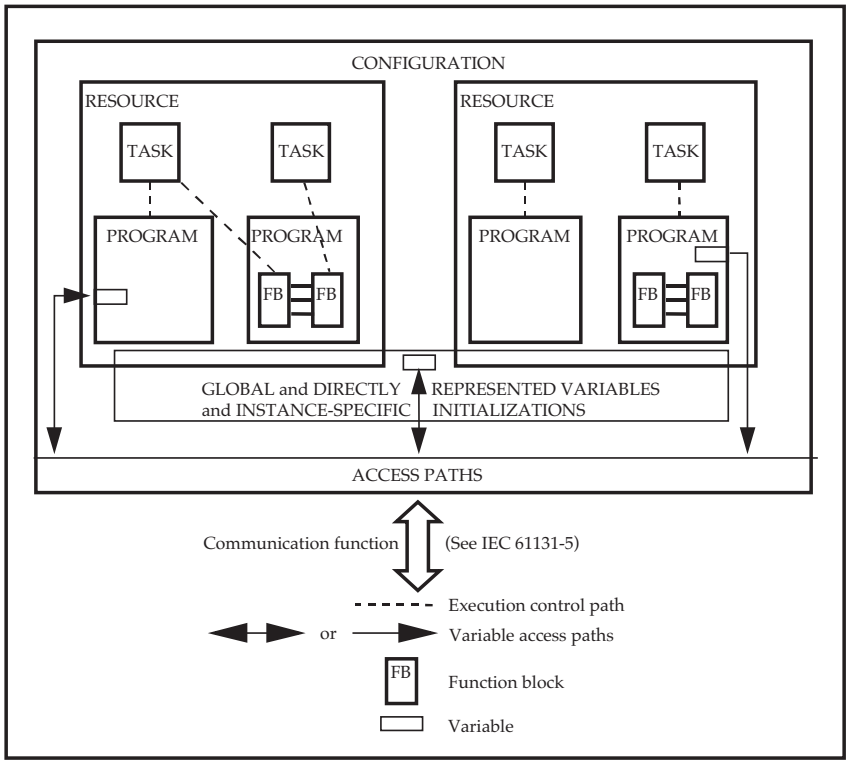


Figure 2.3: The IEC 61131-3 software model.

IEC 61131-1. A configuration contains one or more resources, each of which contains one or more programs executed under the control of zero or more tasks. A program may contain zero or more function blocks or other language elements as defined in this part. Configurations and resources can be started and stopped via the "operator interface", "programming, testing, and monitoring", or "operating system"functions defined in IEC 61131-1. The starting of a configuration shall cause the initialization of its global variables, followed by the starting of all the resources in the configuration. The starting of a resource shall cause the initialization of all the variables in the resource, followed by the enabling of all the tasks in the resource. The stopping of a resource shall cause the disabling of all its tasks, while the stopping of a configuration shall cause the stopping of all its resources. Mechanisms for the starting and stopping of configurations and resources via communication functions are defined in IEC 61131-5. Programs, resources, global variables, access paths (and their corresponding access privileges), and configurations can be loaded or deleted by the "communication function" defined in IEC 61131-1. The loading or deletion of a configuration or resource shall be equivalent to the loading or deletion of all the elements it contains.
In detail, the common elements that compose the software model described above are: (i) *variables* and their data-types, defined according to their visibility and role in the execution of the application. (ii) *Program organization Units* that permit to organize the application according to hierarchical modularity and encapsulation concepts (i.e. POUs can be nested by invocation and declaration). POUs defined in IEC 61131-3 are Programs, functions and Function Blocks. (iii) Configuration and execution control elements, that define the allocation of POUs to the computational resources (processors, tasks) of the application. (iv) *SFC elements*, defined within

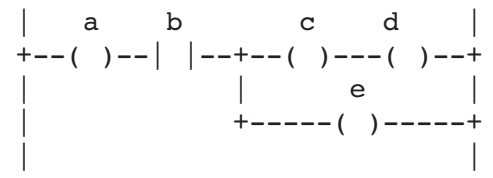


Figure 2.4: A simple ladder diagram example.

common programming features for the reasons described above.

IEC 61131-3 Languages

The definition of IEC 61131-3 programming language have been formalized according to the trends of users and vendors of industrial automation. The internal code of any POU can be programmed with any IEC language, which permit to choose the appropriate implementation for each specific POU, considering performance optimization, type and complexity of operation executed or even background and experience of the programmer. The five IEC languages are : ladder diagram, instruction list, function block diagram, structured text and sequential function chart. The **ladder diagram** is historically the most used notation to implement logic controllers in the industrial automation field. Writing a program is then equivalent to drawing a switching circuit. The ladder diagram consists of two vertical lines representing the power rails. Circuits are connected as horizontal lines, i.e. the rungs of the ladder, between these two verticals. In drawing a ladder diagram, certain conventions are adopted: (i) the vertical lines of the diagram represent the power rails between which circuits are connected. The power flow is taken to be from the left-hand vertical across a rung. (ii) Each rung on the ladder defines one operation in the control process. (iii) A ladder diagram is read from left to right and from top to bottom, the top rung is read from left to right. Then the second rung down is read from left to right and so on. When the PLC is in its run mode, it goes through the entire ladder program to the end, the end rung of the program being clearly denoted, and then promptly resumes at the start. (iv) Each rung must start with an input or inputs and must end with at least one output. The term input is used for a control action, such as closing the contacts of a switch, used as an input to the PLC. The term output is used for a device connected to the output of a PLC, e.g. a motor. (v) Electrical devices are shown in their normal condition. Thus a switch which is normally open until some object closes it, is shown as open on the ladder diagram. A switch that is normally closed is shown closed. (vi) A particular device can appear in more than one rung of a ladder. For example, we might have a relay which switches on one or more devices. The same letters and/or numbers are used to label the device in each situation. (vii) The inputs and outputs are all identified by their addresses, the notation used depending on the PLC manufacturer. In figure 2.4 is depicted a ladder diagram example.

A programming method, which can be considered to be the entering of a ladder program using text, is **instruction lists** (IL). Instruction list gives programs which consist of a series of instructions, each instruction being on a new line. An instruction consists of an operator followed by one of more operands, i.e. the subjects of the operator. In terms of ladder diagrams an operator may be regarded as a ladder element. Each instruction may either use or change the value stored in a memory register. For this, mnemonic codes are used, each code corresponding to an operator/ladder element. The codes used differ to some extent from manufacturer to manufacturer, though a standard IEC 1131-3 has been proposed and is being widely adopted.

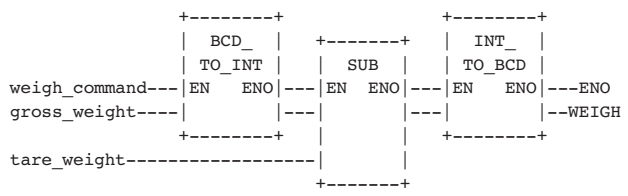


Figure 2.5: A simple FBD example.

An example of IL is reported following

```
ST    Q   ( Store result in Q, i.e. output to Q)
AND   B   (AND B)
LD    A   (Load A)
```

In the first line of the program, LD is the operator, A the operand, and the words at the ends of program lines and in brackets and preceded and followed by are comments added to explain what the operation is and are not part of the program operation instructions to the PLC. LD A is thus the instruction to load the A into the memory register. It can then later be called on for further operations. The next line of the program has the Boolean operation AND performed with A and B. The last line has the result stored in Q, i.e. outputted to Q.

The term **function block diagram** (FBD) is used for PLC programs described in terms of graphical blocks. It is described as being a graphical language for depicting signal and data flows through blocks, these being reusable software elements. A function block is a program instruction unit which, when executed, yields one or more output values. Thus a block is represented in the manner shown in figure 2.5 with the function name written in the box. A function block is depicted as a rectangular block with inputs entering from the left and outputs emerging from the right. The function block type name is shown in the block, with the name of the function block in the system shown above it. Names of function block inputs are shown within the block at the appropriate input and output points. Cross diagram connectors are used to indicate where graphical lines would be difficult to draw without cluttering up or complicating a diagram and show where an output at one point is used as an input at another.

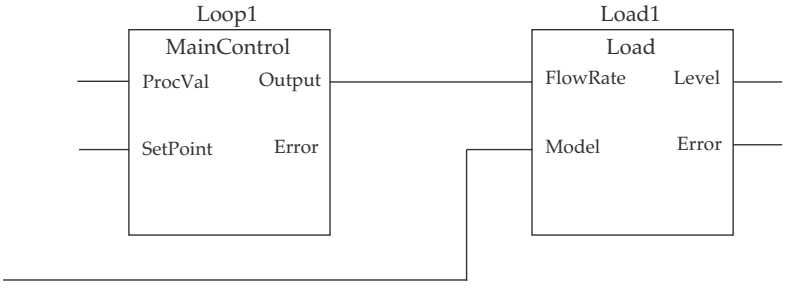
Structured text is a programming language that strongly resembles the programming language PASCAL. Programs are written as a series of statements separated by semicolons. The statements use predefined statements and subroutines to change variables, these being defined values, internally stored values or inputs and outputs. Assignment statements are used to indicate how the value of a variable it to be changed, for example `Light := SwitchA;` is used to indicate that a light is to have its value changed, i.e. switched on or off, when switch A changes its value, i.e. is on or off. The general format of an assignment statement is: `Variable := Expressions;` where Y represents an expression which produces a new value for the variable X.

The **sequential function chart** (SFC) it is not only a language but it is an element for use in structuring the internal organization of a programmable controller program organization unit, written in one of the languages defined in this standard, for the purpose of performing sequential control functions. The definitions in this subclause are derived from IEC 60848, with the changes necessary to convert the representations from a documentation standard to a set of execution control elements for a programmable controller program organization unit. The SFC elements provide a means of partitioning a programmable controller program organization unit into a set of steps and transitions interconnected by directed links. Associated with each

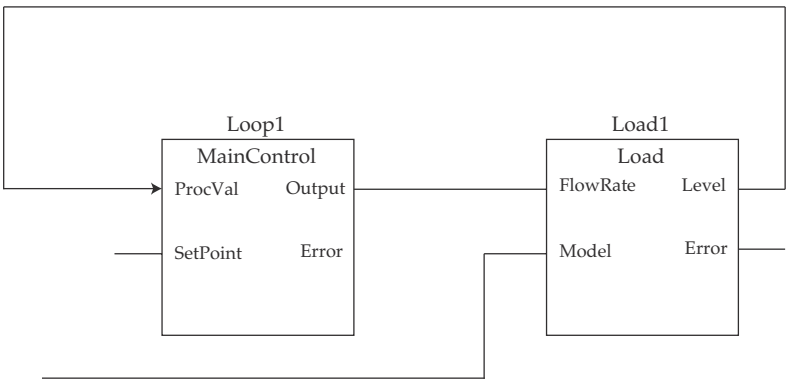
step is a set of actions, and with each transition is associated a transition condition. Since SFC elements require storage of state information, the only program organization units which can be structured using these elements are function blocks and programs. If any part of a program organization unit is partitioned into SFC elements, the entire program organization unit shall be so partitioned. If no SFC partitioning is given for a program organization unit, the entire program organization unit shall be considered to be a single action which executes under the control of the invoking entity. From the syntactical and semantical point of view, Grafcet and SFC are identical with regard to the concepts of activation of steps, enabling of transitions rules of evolution, some example of SFC are shown in figure 3.3 and figure 3.4
To a complete guide to IEC 61131 the reader is referred to [45] and [59].

2.3.2 Standard language: IEC 61499

IEC 61499 provides for the first time a framework and architecture for describing the functionality in distributed control systems in terms of cooperating networks of function blocks. This new standard wants that the benefits of this standard will be understood by a wide audience; including not only people working in industrial control but also those with a general interest in methodologies for modelling distributed systems. In IEC 61131-3 standard there is the definition of function block but it is not possible to use it in distributed systems. There are a number of limitations with the original function block concept introduced by the PLC Languages standard IEC 61131-3. With the IEC 61131-3 Function Block Diagram (FBD) graphical language,



(a) Function block network.



(b) Network with feedback connection.

Figure 2.6: Example of 61131-3 function block connection.

function blocks can be linked by simply connecting data flow connections between block input

and output variables, see figure 2.6(a). Each function block provides a single internal algorithm that is executed when the function block is invoked. The normal execution order is determined by the function block dependency on the other blocks; the order normally runs from left to right because blocks to the right depend on the output values of blocks on the left. However, when a feedback path is introduced, see figure 2.6(b), the execution order cannot be determined from the diagram, since the execution of both blocks depends on an output value of the other block. In a complex network, it is very difficult for a programming system to determine a valid order of execution. To overcome this problem, many IEC 61131-3 programming systems provide additional mechanisms to define the execution order of blocks. For example, the user can view a list of function blocks and manually assign an execution order. Unfortunately, such mechanisms are outside the scope of the IEC 61131-3 standard. As a consequence, an important aspect of a function block network, i.e. the method for defining the execution order of blocks, is not consistent or portable across different control systems. There is one feature in IEC 61131-3 that does provide a crude mechanism for passing execution flow through a chain of function blocks that is worth consideration; that is the use of the EN input and ENO output signals (see fig. 2.7). The EN and ENO signals were intended for function blocks to pass "power flow" when used in

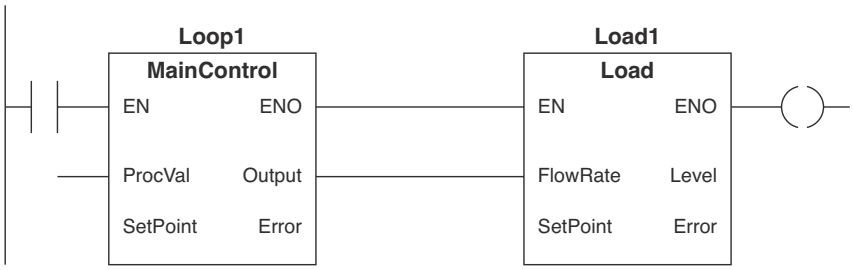


Figure 2.7: Using 61131-3 function block with enable.

rungs of a Ladder Diagram. However, it is now recognised that use of the EN and ENO signals does not provide the degree of flexibility needed for complex FB networks. In effect, the EN and ENO signals can be regarded as a means of passing events between blocks. "EN" signals that the block may be invoked because its input data is ready; "ENO" is signaling that the block has executed and the output data is ready for the next block. We will see that this idea of event passing has been extended in IEC 61499. The focus of the IEC 61131-3 standard (see [45] and [59]) has been to define a software model and languages for PLCs where software is typically running on one processing resource. However, the IEC 61131-3 software model, see figure 2.3, does consider configurations that have multiple resources. The standard provides two different mechanisms for passing data and control signals between resources, namely global variables and communications function blocks.

At the core of the standard is the function block model that underpins the whole IEC 61499 architecture. A function block is described as a "functional unit of software" that has its own data structure which can be manipulated by one or more algorithms. A function block type definition provides a formal description of the data structure and the algorithms to be applied to the data that exists within the various instances. This is not a new concept but based on common industrial practice applied to reusable control blocks of various forms. A good example is the Proportional, Integral and Derivative (PID) block used in many PLCs and controllers. The system vendor will typically supply a type definition for a PID block. The programmer can then create multiple instances of the PID block within the control program, each of which can

be run independently. Each PID instance, such as "Loop1" "Loop2" will have its own set of initialisation parameters and internal state variables and yet share the same update algorithm. IEC 61499 defines several forms of function block to a complete list the reader is referred to ([46], [47] and [60]). The main features of a function block are summarised as follows:

- Each function block type has a type name and an instance name. These should always be shown when the block is depicted graphically.
- Each block has a set of event inputs, which can receive events from other blocks via event connections.
- There are one or more event outputs, which can be used to propagate events on to other blocks.
- There is a set of data inputs that allow data values to be passed in from other blocks.
- There is a set of data outputs to pass data values produced within the function block out to other blocks.
- Each block will have a set of internal variables that are used to hold values retained between algorithm invocations.
- The behaviour of the function block is defined in terms of algorithms and state information. Using the block states and changes of state, various strategies can be modelled to define which algorithms are to execute in response to particular events.

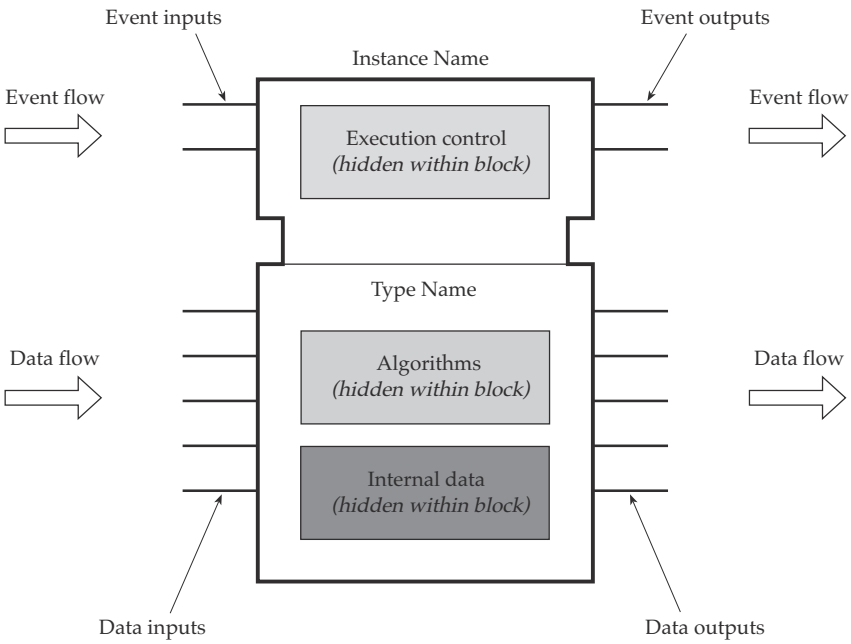


Figure 2.8: 61499 Function block definition.

In figure 2.8, the main characteristics of an IEC 61499 function block are depicted. The top part of the function block, called the "Execution Control" portion contains a definition in some cases, given in terms of a state machine, to map events on to algorithms; i.e. it defines which

algorithms defined in the lower body are triggered on the arrival of various events at the "Execution Control" and when output events are triggered, what the standard calls the causal relationship among event inputs, event outputs and the execution of algorithms. The standard defines means to map the relationships between events arriving at the event inputs, the execution of internal algorithms and the triggering of output events. The lower portion of the function block contains the algorithms and internal data, both of which are hidden within the function block. A function block is a type of software component and, if well designed, there should be no requirement for a user to have a detailed understanding of its internal design. A function block relies on the support of its containing resource to provide facilities to schedule algorithms and map requests to communications and process interfaces. The standard states that a resource may optionally provide additional features to allow the internals of a function block to be accessed. So there may be "backdoor" methods to access function block internals; however, from the IEC 61499 architecture view point, control variables and events are only passed by the external exposed interfaces.
An another important concept in IEC 61499 is the ability to define a function block type that defines the behaviour and interfaces of function block instances that can be created from the type. This is synonymous with the way in object oriented (OO) software that the behaviour of object instances is defined by the associated object's class definition. A function block type is defined by a type name, formal definitions for the block's input and output events, and definitions for the input and output variables. The type definition also includes the internal behaviour of the block but this is defined in different ways for different forms of block.
The behaviour of a basic function block is defined in terms of algorithms that are invoked in response to input events. As algorithms execute they trigger output events to signal that certain state changes have occurred within the block. The mapping of events on to algorithms is expressed using a special state transition notation called an *Execution Control Chart* (ECC). The internal behaviour of composite function block and subapplication types is defined by a network of function block instances. The definition therefore includes data and event connections that need to exist between the internal function block instances. To a complete guide to definition of different function block type and to a complete description of IEC 61499 the interesting reader reader is referred to [46], [47] and [60].

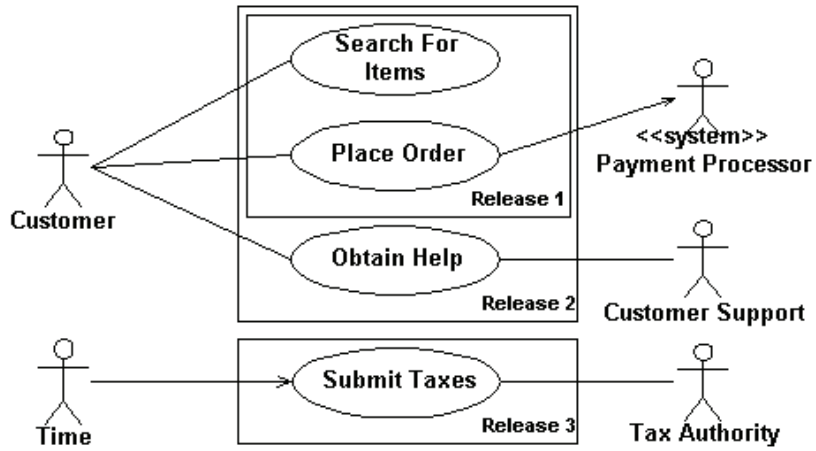
2.4 Object Oriented

One of the most important goal of software engineering is to reduce the system complexity. Object-oriented programming (OOP) is a programming paradigm that uses "objects" (data structures) consisting of data fields and methods together with their interactions to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOP.
An object is actually a discrete bundle of functions and procedures, all relating to a particular real-world concept such as a bank account holder or hockey player in a computer game. Other pieces of software can access the object only by calling its functions and procedures that have been allowed to be called by outsiders. A large number of software engineers agree that isolating objects in this way makes their software easier to manage and keep track of. However, a not-insignificant number of engineers feel the reverse may be true: that software becomes more complex to maintain and document, or even to engineer from the start. Object-oriented

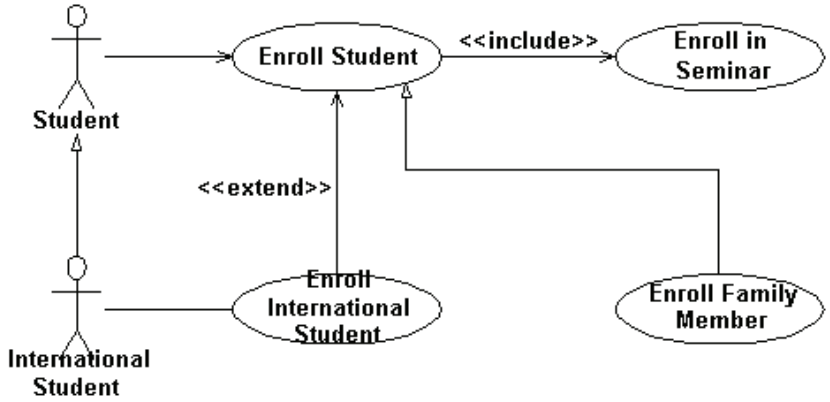
programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic[citation needed]. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming which had been dominant for many years that focused on the function of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines). This more conventional approach, which still persists, tends to consider data and behavior separately. An object-oriented program may thus be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. The use of OOP and standardized languages, such C++ and java, is well-estabilished in the domain of business software and is constantly growing in the programming of real-time embedded systems. In next section we will see the UMl language borns in software engineering filed and apply in industrial automation

2.4.1 UML - Unified Modeling Language

In last years there was an increase of methods and notations within the paradigm of OO software engineering have had the result of confusing users and software developers. In order to supply to the absence of standardize graphical symbols to model OO specification, a consortium of the major software companies and analyst, the *Object Management Group* (OMG, see [66] and UML2), defined a standard set of unified notations, covering all the aspects of analysis and design of OO software systems, which is called *Unified Modeling Language* (UML) (to a complete description of the language see [77]. It is important to note that it does not underline any specific methodology, but gives only the modeling language that may help in the development of OO analysis and design activities. In particular, it defines the syntax of eight different diagrammatic notations and explains with informal description and examples their meaning with respect to the OO paradigm. These notations are called: *Use Case Diagrams, Class Diagrams, Collaboration Diagrams, Sequence Diagrams, State Diagrams, Activity Diagrams, Component Diagrams* and *Deployment Diagrams*. A Use Case Diagram specifies the functional requirements of the global system or of one of its sub-systems, similarly to a Context Diagram as used in State Activity and state Diagram: the boundaries of the system are denoted by a square box in which the required functionalities, called "use cases" are drawn as ovals. External entities, called "actors" are in general depicted with stick human figures, which recalls the fact that a typical software system interacts with a human user. However, in real-time and control systems, physical devices and controlled processes can be considered as external actors. Use cases can be hierarchically reined or related to each other, by meal's of "Relationship arrows" for which the UML standard gives some specialized stereotypes: <<extend>> <<use>> and <<include>>. Some examples are reported in figure 2.9(a) and figure 2.9(b), taken from (union), show, respectively, art example of Use Case Diagram and an example of use case relationships. When applying Use Case Diagrams to model functional requirements for real-time systems it may be useful to associate with links between actors and the system



(a) An example of UML Use case diagram.



(b) An example of UML Use case relationship.

Figure 2.9: Example of UML diagram.

a list of events which may stimulate system responses. Moreover behavioral aspects related to a particular use case may be specified with the help of Sequence Diagrams or Statecharts. However, it should be noted that use case modeling does not imply any object identification strategy, but the behaviour is intended with respect to the system in its entirety. For teh description of the other class diagram the reader is referred to [77]
The aim of UML language is to model a system with a graphical approach, for systems it means not only a "physical system" but also a software, a communication protocol etc. UML born in software engineering and two important extension to help the designer to model real time systems and physical systems. The benefits of object-orientation are nowadays more and more recognized also by real-time software developers, with the consequence that an increasing number of engineering methods for the analysis and design of real-time and control systems, based on UML and successfully applied to practical cases thanks to the help of specifically designed CASE tools and extensions to the basic UML notation. With regard to the extensions of UML, these are in general introduced because several critical aspects of real-time systems have been recognized to require explicit arid peculiar support by the specification language. In fact, the analysis of objects and their interactions within a real-time system lead to the definition of very different models compared to those designed in the business domain.

The UML standard explicitly refers to active objects in the description of Collaboration and Sequence Diagrams and to events in the description of diagrams for the behavioral specifications, mainly Statecharts and Sequence Diagrams, it does not define how to specify the details of inter-task communications, signal/event-based interfaces and so on. UMl provides a powerful extension mechanism, based on the fact that the language is defined in terms of *meta-model*. Thanks to the meta-model, it is possible to introduce and integrate in the modeling framework domain-specific concepts otherwise difficult to specify. In particular a new class of the UML meta-model can be beaned as sub-class of one of its modeling elements. The new class, called stereotype, may have: (i) *constraints*, expressed in the Object Constraint Language that define well-formedness rules that should be satisfied by correct models in which the stereotype is used. (ii) *Tagged values*, that represents properties and information that an instance of the stereotype may have, but are not possible to define in the basic element of the meta-model. An interesting UML profile for real-time systems is the one that have beefs derived by the notation defined in the *Real Time bedtime Object-Oriented Methodology* (ROOM) (see [83] and [82]). The major extension of UML-RT are related to the introduction of constructs to specie highly encapsulated concurrent and event-driven modules togher with their well-defined signal interface.
To have a better modeling of physical system it borns SysML (see [1] for more details), an open source specification project founded by the "SysML Partners" in 2003, that satisfies the requirements of the UML for Systems Engineering RFP (Request For Proposal). It allows to model systems which are composed by software and hardware components. More precisely, SysML is a strict profile of UML 2.0 which is developed to deal with the systems issues. SysML introduces the *Block Definition Diagram (BDD)*: in this diagram blocks are the basic structural elements in SysML; they can be used to represent hardware, software, facilities, personnel, or any other system element. BDD is a static diagram which describes the blocks in a system and the different static relationships between them. The latter elements can represent dependence, generalization, association, aggregation and composition.

2.5 Conclusions

In this chapter a brief overview on control design in industrial automated systems and how the modern software engineering tecquiniques, like object orienting programming, are applied to modern industrial automated systems is reported. The first step of this chapter is define what is a "design pattern"in industrial automation. The design patterns in automation have been adopted from the object-oriented programming patterns, this reflects the generic and reusable nature of the original object-oriented patterns. In general, the context of automation seem to affect only on more detailed, or more context-specific levels, to the architecture of object-oriented software. Two example of design pattern used in industrial automation are the standard ISA-88 (see [15], [49] and [50]) and GEMMA (see [65] and [19]).
The approach of standard ISA-88 is applied to production process, and this standard wants to help the designer to separating the design of control systems in a hierarchical approach. This hierarchical approach is implemented defining two level, an high level where there is a sequence of actions (recipes) and the low level where this actions are implemented (product equipment capability). This standard do not help the designer to choose how to divide the systems, this part is left to the ability of single designer without define a *standard path*. In the low level of this architecture is implemented also the part of control logic devoted to check safety condition, this means the low level has to know the condition of high level and the condition of the other part of the low level. With this "distributed information" in the low level there is

Feature	Function Blocks	Objects	Comments
Encapsulatated data	✓	✓	Object may contain data that is also instance of the other objects. Function blocks may contain instances of other function blocks.
External interface	✓	✓	In IEC 61499 function blocks, there is not distinction between private and public interface.
Invocation	Function blocks use input and output variables and events.	Objects have methods with arguments and returned values.	With function blocks, data can be synchronised with an event.
Inheritage	✓	×	Currently in IEC 61499 there is no mechanism for a function block to inhert behaviour.
Polymorphism	✓	✓	IEC 61499 introduce a new adaptor concept that allows function blocks to share common interface.
Instantiated from a class	Function blocks instances are defined from functions block type.	Objects have methods with arguments and returned values.	

Figure 2.10: Comparison between objects and function blocks.

no information encapsulation and this involves a low modularity. Another important point is S88 do not define not establish a formal standard states but only a set of possible set, leaving to single designer the choice of the set and the command operation. GEMMA is another design pattern applied usually in manufacturing systems. It is based on the allocation of the functionality of the entire machine on standard state operation. With this allocation GEMMA wants to help the designer to build in a structured design approach the control logic. This allocation helps also the designer in check all the condition working and fault condition, but it is not clear how allocate the different part of control logic in the state, and how to divide the different part of control logic.
In industrial automated systems the preferred options to programming are the language defined in the standards IEC 61131-3, and IEC 61499. from a first comparison the function blocks of IEC 61499 are n many way similar to objects, a complete comparison between objects and function blocks are depicted in figure 2.10. The polymorphism and inheritance are very strongly related to object oriented programming. They provide the means to dynamically create, destroy and bind objects at run-time. The IEC 61499 does not include full power to create dynamic behavior of object-oriented software. However, in IEC 61499 it is possible to find some patterns for distributed control logic, in fact the standard helps the designer to allocate the task on the distributed resources.
The software to design control logic for automated systems shows low characteristics of reusability and modularity. These characteristics are strongly emphasized and realized in the so-called object-oriented methodologies introduced in computer-science area since a long time. Recently, this approach is fruitfully pervading the industrial automation world too in order to become one of the means to reduce the complexity and to introduce standardization in software design. Industrial informatics is aimed at helping the designer introducing standardization giving some methodologies able to guide the designer towards the modular and reusable software. In this field, IEC 61131-3 identifies standard programming languages for PLC systems ([59] and

[45]), mainly focusing on centralized computing platforms. Differently, the IEC 61499 standard (see [60], [46], [47] and [92]) enriches the Function Block framework of IEC 61131-3 considering explicitly a distributed network of computing elements (usually organized using peer-to-peer paradigm). This allows a simpler decomposition of the whole automation logic control in many smaller objects (even for centralized implementation). Among the different strategies for logic control proposed in literature, in [93] the IEC 61499 framework is exploited to define a formal modeling suitable for verification, while in [10] and [8] the Mechatronic Object has been introduced to deal with mechanical and electronic issues involved in the automation of industrial plant. Another class of methodologies that is important to cite is the so called MDA (Model Driven Approach) (see [66] and [54]) where general modeling languages (such as UML) are exploited to model systems independently of their implementation platforms, to introduce agent-based approaches in control design and to generate platform-specific software code automatically from high level models (see [4], [26] and [25]).
In the next chapter will show a new approach to define an hierarchical architecture based on the new concept of the *Generalized Actuator*

Chapter 3

Architecture in industrial automation: The Generalized Actuator approach

In this chapter an effective design approach to the design of control logic in industrial automated systems using hierarchical control architectures is presented. The main characteristic of the solution is the clear and structural separation between "policies" and "actions" deriving from the use of a novel abstract entity in modelling automation plants: the *Generalized Actuator*. Particular attention is paid to illustrate how to define generalized actuators starting from a "bare plant". The potentialities and advantages deriving from this methods of this method are emphasized by means of some illustrative case study.

3.1 Introduction

In nowadays manufacturing applications, high complexity automatic machineries are employed to accomplish production tasks; designing, coding and testing logic control software for such machineries is a challenging task, especially considering that such software embed different functionalities as: logic control in nominal conditions, diagnostics, fault reconfigurations, safety functions, quality control etc. In chapter 2 it was presented a state of the art on industrial automation software engineering, concepts as modularity, encapsulation, composability and reusability and are strongly emphasized and profitably realized in the so-called object-oriented methodologies. These methodologies are fruitfully pervading the industrial automation world too, as testified not only by current availability of commercial products conforming, at least to a certain extent, to the standards defined for this specific domain by International Organisms, such as IEC and OMG, but also by some interesting proposal about generally applicable modelling and design frameworks recently published in the scientific and technical literature. A promising approach to reduce complexity and introduce standardization is to exploit classical concepts used in software engineering as object oriented programming. Basically the focus should be posed on the identification of design patterns to guide the control engineer in the

control software design procedure; such design patterns should be obtained by a right mixing of software engineering machineries (UML, encapsulation...), automation standards (e.g. IEC 61131, IEC 61499) and theoretical machineries (DES theory). The current trend in industrial automation, as testified also by the available commercial products, is therefore to exploit the so-called *object oriented methodologies*, which are well known in software engineering field. From the latter point of view, in [94], [93] and [92], the framework of IEC61499 is exploited to define a formal modelling suitable for verification. In [11], [12] the *Mechatronic Object* has been introduced to deal with mechanical and electronic issues involved in the automation of industrial plant. This approach has been further extended in [13], [8] and [9] where a solid unification of dynamic systems and industrial control software modelling is proposed. In [86], [87] and [88] a model integrated paradigm is introduced to represent mechatronic systems. Differently, in ([31]) the *Control Module* is introduced following the agent paradigm to achieve a modular representation of automation functions for flexible manufacturing systems. The interested reader can find exhaustive information on the cited methodologies in [82], [45] and [17] and references therein, while their exploitation in industrial automation is described.
As a matter of fact, a key element for the effectiveness of a proposed modelling framework is the correlation with a clear procedure to deal with it. Taking inspiration from this basic consideration, the main focus of this paper is to present a *modelling framework* and a *design procedure* to realize automation functions exploiting a clear and structural separation between *Policies* and *Actions*. Toward this purpose, a novel entity is introduced for modeling industrial automation systems: *the Generalized Actuator* (GA). The main characteristics of the proposed modelling framework and design procedure are the following:

- Introduce a straightforward way to encapsulate “actuation mechanisms”, using GA;
- Effectively support hardware virtualization, component interoperability and reusability;
- Allow hierarchical management of a plant, separating control policies from actuation mechanisms;
- Allow detection of anomalous situations following a distributed hierarchical approach.

3.2 Classic design procedure

To introduce the design procedure, let us start presenting a simple example: a drilling machine in a manufacturing system. This example is an adaptation of the drilling machine in processing station of FESTO manufacturing systems explained in appendix B. We will deal with the different functionalities desired for the system in different steps; namely these functionalities are:

1. To drill a workpiece using a presence sensor;
2. Control duration of drillin according to a suitable policy;
3. Diagnostic sensors faults;

The system is schematically depicted in figure figure 3.1; it is composed by a rotary table that feeds some workpiece to a drilling station that perform the drill operation over the workpiece. The rotary table is actuated through the command signal `RotaryTable`; the system is equipped with a sensor that indicates when a workpiece is in the correct position to be drilled

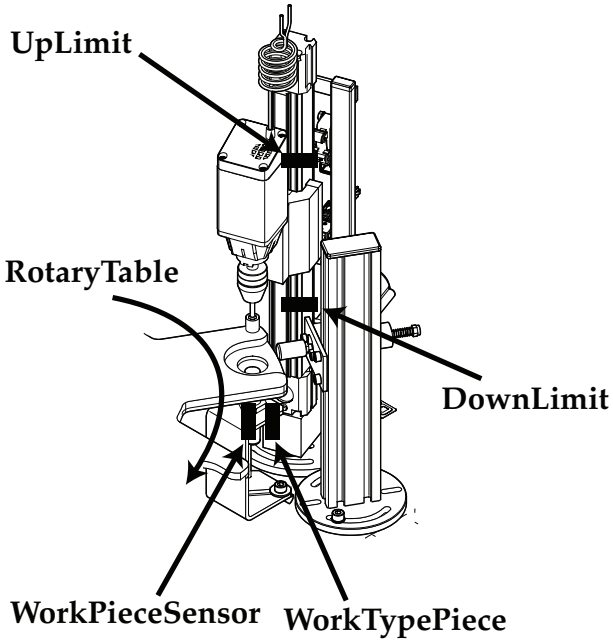


Figure 3.1: Drilling module of Festo FMS.

(signal `WorkPieceSensor`). The electric motor that vertically moves the drilling unit has two direction of movement, decidable through command signals `DrillingUP` and `DrillingDOWN`; the combination `DrillingUP=1` `DrillingDown=0` causes the upward direction while the combination `DrillingUO=0` `DrillingDOWN=0` causes the downward direction. Two sensors indicate the up limit stop (signal `LimitUP`) and the down limit stop (signal `LimitDown`) of the drilling unit. The drilling unit is equipped with a drilling tool mounted into a spindle moved by an electric motor; the spindle has two different direction of movement, decidable through command signals `DrillingRotationON` and `SpindleMotorDirection`, namely clockwise movement can be issued through the combination of `DrillingRotationON=1` and `SpindleMotorDirection=1`, while the anticlockwise movement can be issued through the combination of `SpindleMotorON=1` and `SpindleMotorDirection=0`. When a new workpiece arrives under the drilling unit, this must reach its downward position and the spindle must turn clockwise to perform the drilling operation to the workpiece. The drilling operation must continue for five seconds. After this time interval, the drilling unit must reach its upward limit while the load is expelled; during this operation the spindle must turn anticlockwise to allow the correct extraction of the drilling tool from the load. The overall process must start when the command `StartProcess` is active and should stop when `StartProcess` becomes false. In table 3.1 a description of signals used in the example is given.

To solve the considered problem the workpiece presence sensor is assumed to be ideal considering that the signal `WorkPieceSensor` immediately rise only when the load is in the correct drilling position and all the workpiece are drilled for the same time. In figure 3.2 the "common"SFC solution is reported and it reflects the usual approach adopted in industrial automation design. The SFC diagram perform all the action described before but the functioning logic behind the overall process is hidden in the graph and it is impossible to distinguish between the implementation of the main functions of the system. As a matter of fact, despite the use of a graphical language, the designed solution lacks of separation between logic policies

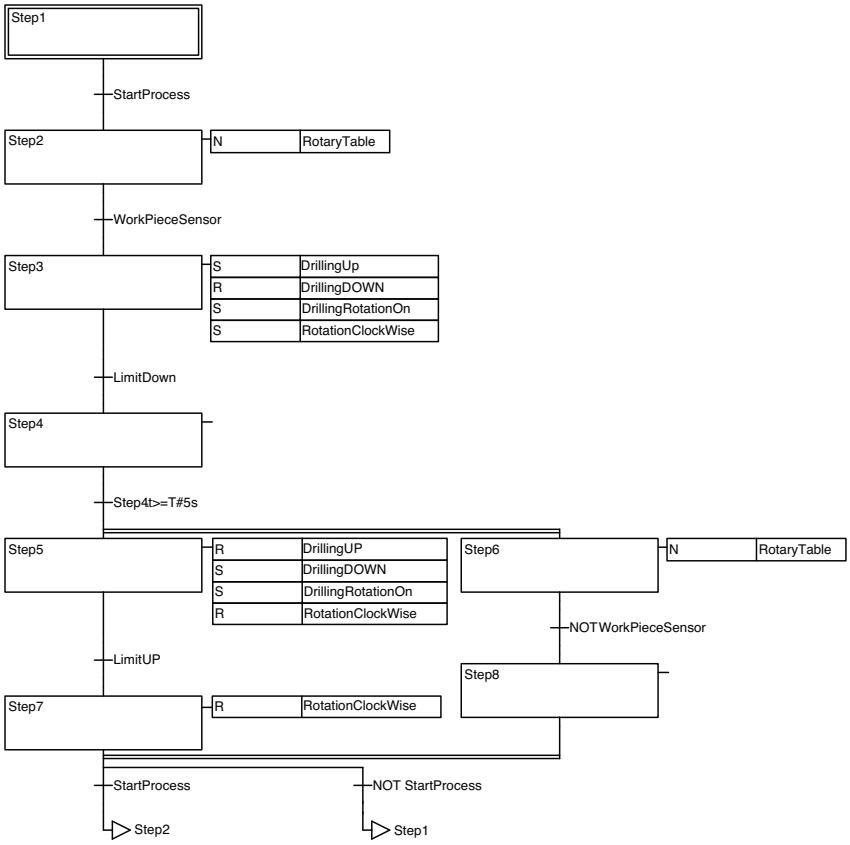


Figure 3.2: "Common" SFC solution for the case in example

and actuation mechanisms, reflecting into a lack of reusability and modularity; this directly affect the readability of the software but also (and especially) the possibility to make some changes quickly and easily as it can be noted considering the following modifications to the plant and policies.

1. Suppose that the considered system is equipped with a presence sensor which cannot be considered as ideal: signal `WorkPieceSensor` becomes true as soon as the workpiece reaches the sensor but this position is not correctly centered below the drilling unit; the belt must therefore move for a given time interval that depends on its actual speed and the load dimensions in order to bring the load in the correct position.

2. Suppose that the system can manage two different types of workpiece and, depending by the kind of workpiece (indicated by signal `WorkPieceTypeSensor`) the drilling operation must be three or five seconds long (i.e. the reference for the temperature control changes according to the actual product).

The control logic for this more involved situation is decipted in figure 3.3; it's possible to note that the new solution is slightly different from the starting one depicted in figure figure 3.2. But it is more interesting noting that the modification (1) is related to an action sensor (generally to an actuation mechanism) while modification (2) is referred to a policy change but this characteristics are not clearly distinguishable in the SFC diagram (see figure figure 3.3 with the highlighted changes). This example testifies that in the classical design approach there is a

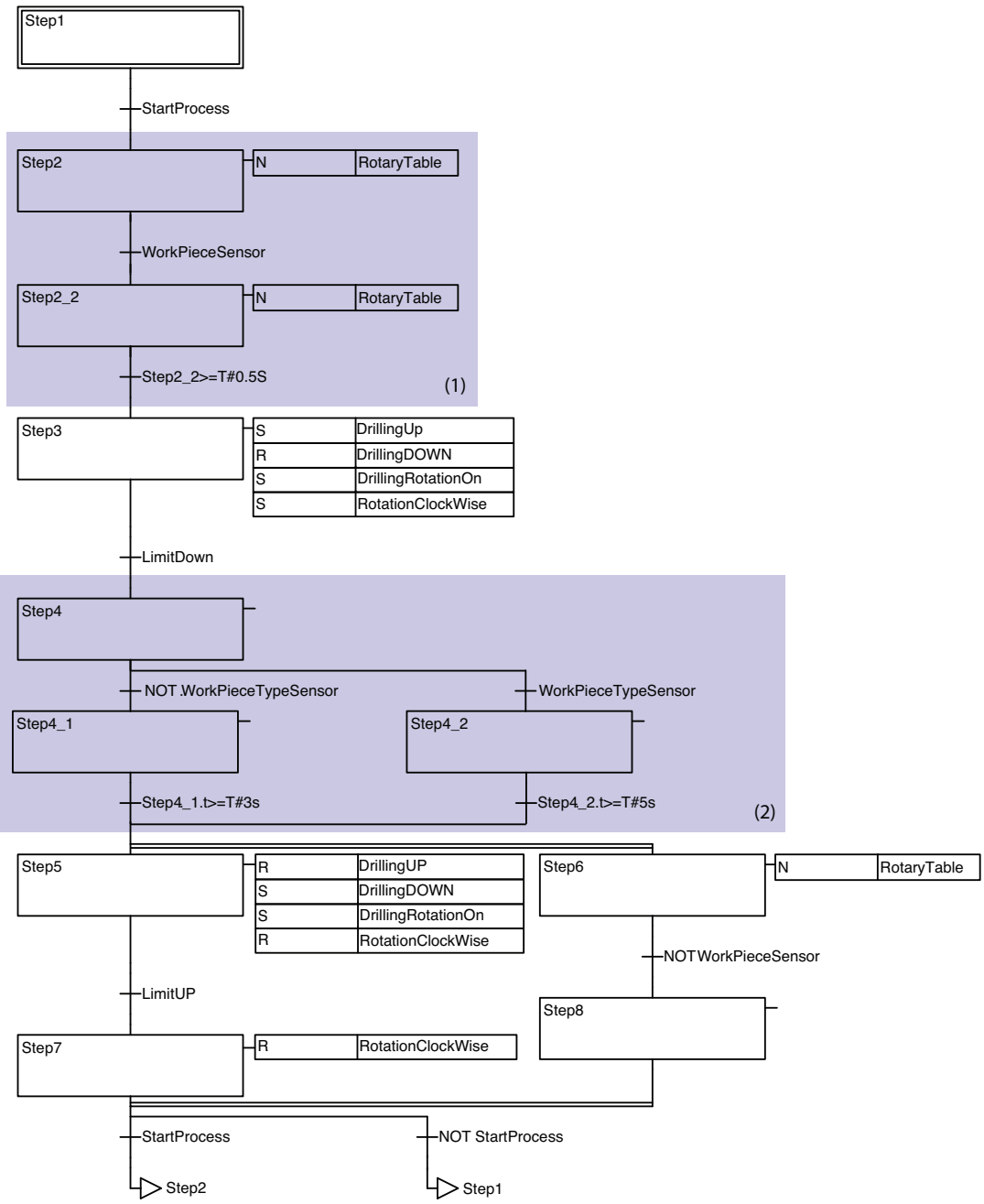


Figure 3.3: "Common" SFC solution for the case in example with new specification.

Signal	Meaning	Type
StartProcess	Operator command to start the process	Digital
RotaryTable	Command signal to move the rotary table	Digital
WorkPieceSensor	Workpiece presence sensor readings	Digital
WorkPieceTypeSensor	Workpiece type sensor readings	Digital
DrillingDown	Command signal to move the drilling unit down	Digital
DrillingUp	Command signal to move the drilling unit upt	Digital
LimitUP	Drilling upward limit signal	Digital
LimitDOWN	Drilling downward limit signal	Digital
DrillingRotationON	Command signal to move the spindle	Digital
SpindleMotorDirection	Command signal to motion direction for the spindle	Digital

Table 3.1: List of signals used in the drilling example

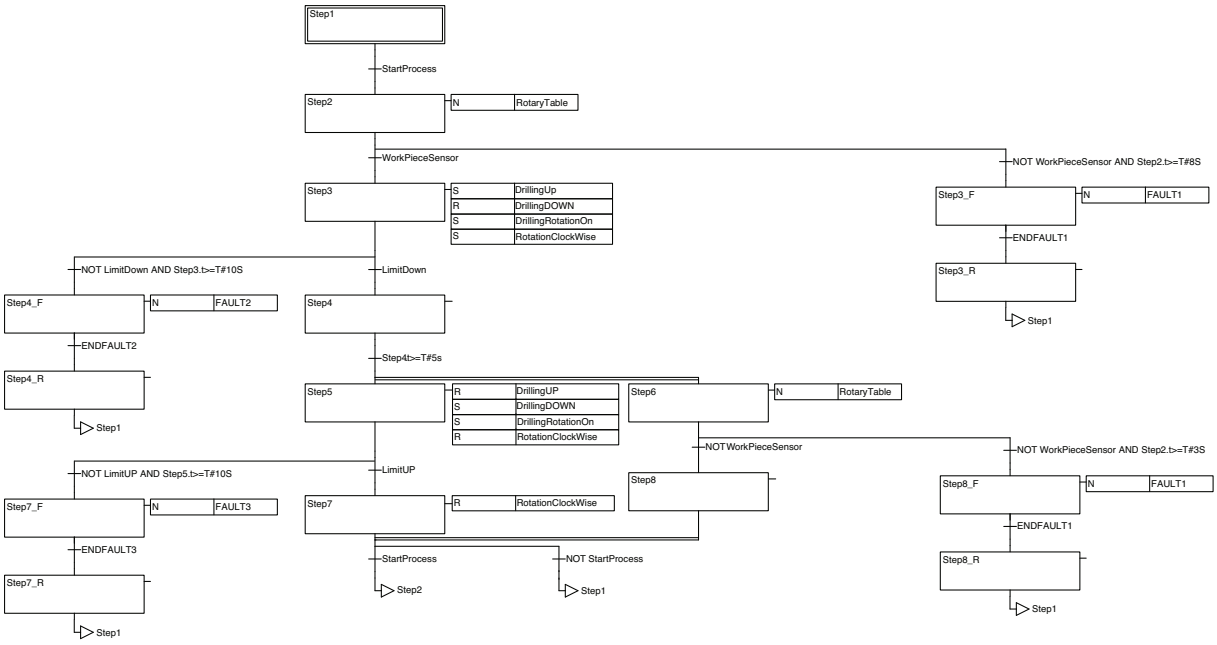


Figure 3.4: "Common" SFC solution for the case in example with fault.

mixing between mechanisms and policies and testifies also how this mixing minimize both the modularity and the reusability of the control software. The lack of reusability can be explained in this way: in the case in the drilling station of the example must be used in another system but in a different mode, it is clear that the software designed shown in figure figure 3.2 or figure 3.3 cannot be reused (typically) without some modifications. Generally, all the control logic are designed for each machine because the adaptation of parts of existing software is difficult in industrial automated systems.

Another worthy remark concerning the solution of figures figure 3.2 and figure 3.3 is to note that the diagnostic phase has been completely disregarded. Even diagnosis of anomalous situations can be considered at two different levels: detection of mechanisms failures (e.g. sensor or actuator faults, components malfunctioning, etc.) and functional anomalies (e.g. forbidden control sequences that occur due to external influences). If we consider possible faults on sensors or actuators on case of the drill unit we can define a control logic as in figure 3.4, while if we consider the faults on drill unit with the modification (1) and (2) we can define a control logic as in figure 3.5. If we compare the starting solution (see fig. 3.2) and the solution with faults

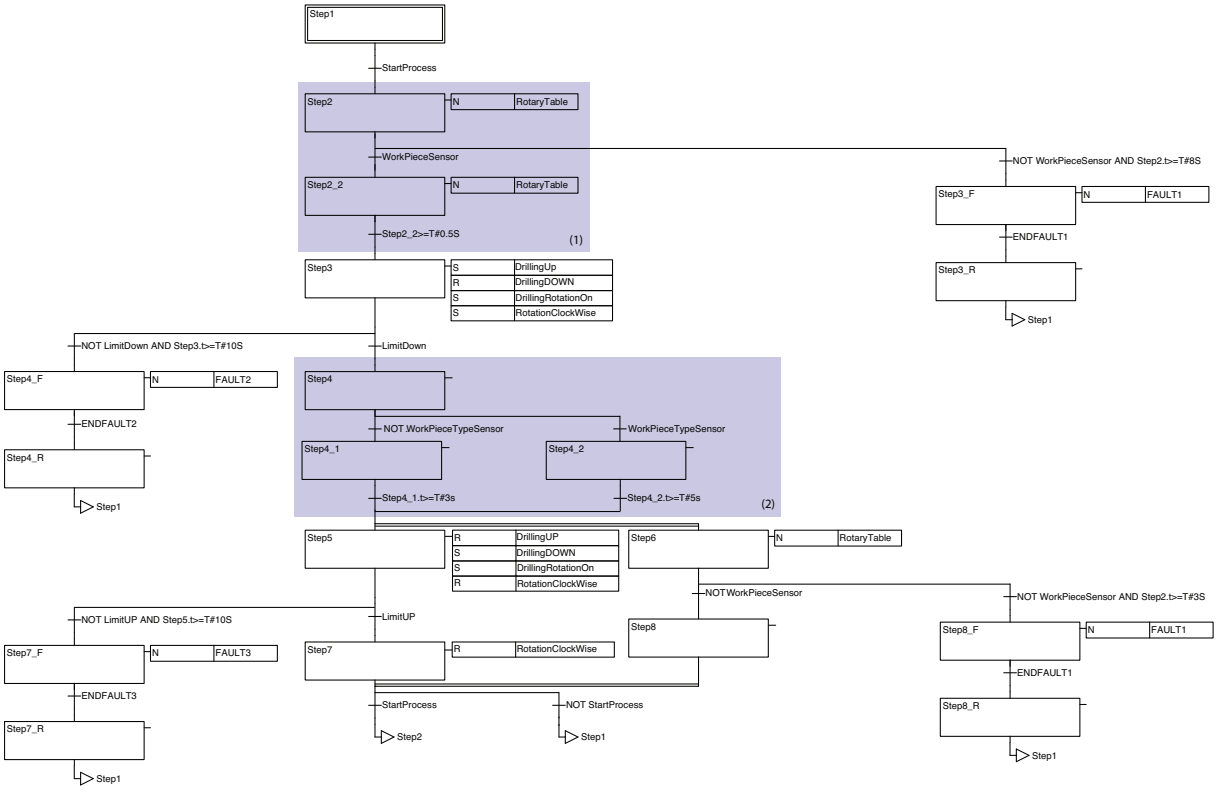


Figure 3.5: "Common" SFC solution for the case in example with new specification and fault.

(see fig. 3.2) it is important to note how add some simple specification the control logic has an explosion of complexity. Another note is regarding that a control logic as the one presented in figure figure 3.2 and figure 3.3 prevents from obtaining a separation between mechanisms diagnostics and policies diagnostics; then also for diagnostic software are preferable characteristics of modularity and reusability especially when it is required the system to be tolerant to faults having to distinguish between mechanisms reliability (e.g. considering redundancy) and policies reconfigurability (e.g. switching between different control logic).

3.3 Generalized Actuator approach

The novel approach proposed in this thesis starts from the idea of considering logic control as a recipe mainly composed by two ingredients: (i) a set of basic actions, (ii) one or more desired sequences to coordinate actions execution. The first ingredient represents mechanisms of functionality implementation, while the second represents the control policy. As enlightened previously the needs for reusable, modular control software require the two to be completely independent. First of all, all the action/mechanisms are defined, using the GA entity; afterwards the overall control policy is considered.
The first step for mechanism definition is to identify basic actions that cannot be reasonably furthermore decomposed. For the considered system the basic actions to perform are (1) move the workpiece in drilling position, indicated as `Positioning`; (2) expel the drilled workpiece,

indicated as `Expulsion`; (3) move the drill unit upward, indicated as `DrillGoUp`; (4) move the ram downward, indicated as `DrillGoDown`; (5) control the spindle rotation, indicated as `SpindleControl`.
Each basic action is then associated with a set of actuators and sensors that physically perform the action. As depicted in figure 3.6, action Positioning and `Expulsion` both involve sensor `WorkPieceSensor` and actuator `RotaryTable`. Actions `DrillGoUp` and `DrillGoDown` involve sensors `LimitUP` and `LimitDOWN`, respectively, and actuators `DrillUP` and `DrillDOWN`, both of them. Finally action `SpindleControl` involves actuators `DrillingRotationON` and `SpindleMotorDirection`. At this point, the proposed subsequent step is the effective

Actions	Sensors	Actuators
Positioning	WorkPieceSensor, WorkPieceTypeSensor	RotaryTable
DrillGoUP	LimitUP	DrillUP, DrillDOWN
DrillGoDOWN	LimitDown	DrillUP, DrillDOWN
Expulsion	WorkPieceSensor	RotaryTable
SpindleControl		DrillingRotationON, SpindleMotorDirection

Actions	Sensors	Actuators
Positioning	WorkPieceSensor, WorkPieceTypeSensor	RotaryTable
DrillGoUP	LimitUP	DrillUP, DrillDOWN
DrillGoDOWN	LimitDown	DrillUP, DrillDOWN
Expulsion	WorkPieceSensor	RotaryTable
SpindleControl		DrillingRotationON, SpindleMotorDirection

Actions	Sensors	Actuators
Positioning	WorkPieceSensor, WorkPieceTypeSensor	RotaryTable
DrillGoUP	LimitUP	DrillUP, DrillDOWN
DrillGoDOWN	LimitDown	DrillUP, DrillDOWN
Expulsion	WorkPieceSensor	RotaryTable
SpindleControl		DrillingRotationON, SpindleMotorDirection

Actions	Sensors	Actuators
Positioning	WorkPieceSensor, WorkPieceTypeSensor	RotaryTable
DrillGoUP	LimitUP	DrillUP, DrillDOWN
DrillGoDOWN	LimitDown	DrillUP, DrillDOWN
Expulsion	WorkPieceSensor	RotaryTable
SpindleControl		DrillingRotationON, SpindleMotorDirection

Figure 3.6: Actions, sensors and actuators of the systems.

definition of GAs following this basic concept (which is a sort of definition of a GA): every GA is a "virtual"actuator with the following characteristics:

- it is in charge of the execution of a small subset of the basic actions identified in previous

steps (hence it handles actuators and sensor associated to them);

- it is *always "alive"* during the operations of the automation plant, even if no specific action is required to it.

In order to give effective "guide-lines"for this phase, the following rules are introduced:

- the union of the actuators associated with the set of GAs must be equal to the whole actuators set of the system (for what concern the sensors usually the same condition should be satisfied but it is not mandatory);

- pursuing a non interference idea, sets of sensors and actuators belonging to different GAs must be disjointed.

In the considered example, the situation depicted in lower part of figure 3.6 is obtained. Looking for common equipment used in different actions, leads to group them in three GAs respectively devoted to the drilling workpiece, move the drill unit and control spindle rotation control.
From the considered example, it is immediate to note that there exist two different kinds of actions and, consequently, of GAs; there are actions which structurally terminate after a finite time (e.g. action `Positioning` implies moving the belt until the workpiece reaches the drilling position), while there are others which, in principle, could continue for an infinite time and whose termination has to be decided "externally"(e.g. action `SpindleControl`).
The GAs associated to the first kind of actions are denominated **Do-Done** GA. They are characterized by a input signal `Do` used to command the starting of an action, an input signal `DoWhat` to specify what kind of action has to be performed (if more than one is available) and an output signal `Done` to signal when the action has terminated successfully.
Differently, the GAs associated to the second kind of actions are denominated **Start-Stop** GA. Their characteristic I/O signals are the input `Start` to command the beginning of an action, defined by the input `StartWhat`, and the input command `Stop` to stop the action.
For the presented example, the following GAs can be defined:

- `WorkPieceMotion`, a Do-Done GA that is devoted to workpiece positioning;
- `DrillMotion`, a Do-Done GA that is aimed at moving the drill unit;
- `Spindle`, a Start-Stop GA that is aimed at controlling the spindle rotation of drilling.

These three GAs are described and realized using the Function Block (FB) formalism defined in IEC61131-3 (see appendix B to see the code). It is worth noting that the FB are not used by chance, in fact they represent Program Organization Units (POUs) which have to be always active during overall control execution.

3.4 Generalized actuator definition and design procedure formalization

We are now ready to define the structure of a GA specializing its input/output interface and its basic dynamics by means of a state diagram; after that, generalizing the procedure proposed in section 3.3 to solve the benchmark example, we furnish some guidelines to design the control logic using the GA approach.

In Figure 3.7 the interface section of both Do-Done GA (Figure 3.7(a)) and Start-Stop GA (Figure 3.7(b)) are depicted; in both cases the interface can be mainly divided in two sections in the following described.

1. **Interface to policy**: this section represents the input/output section between the GA and the supervision policy. It can be further decomposed in two subsections separating the standard communications between the GA and the policy and all the case dependent communications.

   Standard interface: embeds all command inputs for the GA and the outputs that communicate the actual state of the GA and the task that it is accomplishing. More in details the Do-Done GA will receive as command the `Do` signal to start operations and the `DoWhat` signal to specify the desired action, while the Start-Stop GA will be commanded through inputs `Start` to start operations and `Stop` to conclude operations, and through signal `StartWhat` to define the required action. In both cases input signals `Alarm`, `AlarmType` can be used to communicate to the GA the occurrence of an external anomalous situation. The outputs of this section are, for the Do-Done GA, the `Done` signal by which the GA communicate that the task has been performed and the `DoneWhat` signal by which the terminated task is specified; the Start-Stop GA outputs are the signal `DoingWhat` representing the task that the GA is performing. In both kind of GAs, a `State` signal communicate the actual state in which the GA is evolving.

   Communications: represents all the non standard communications between the policy and the GA, as the results of sensor readings filtering (e.g. the `WorkPieceTypeSensor` signals in `WorkPieceMotor` GA, that distinguish between two different kind of worpiece filtering the sensor readings `WorkPieceTypeSensor`.

2. **Low level interface**: this section contain all the interfaces with the low level world; even this section can be further decomposed in two sub sections considering the constant parameters used by the GA separated from the physical interconnection with the plant.

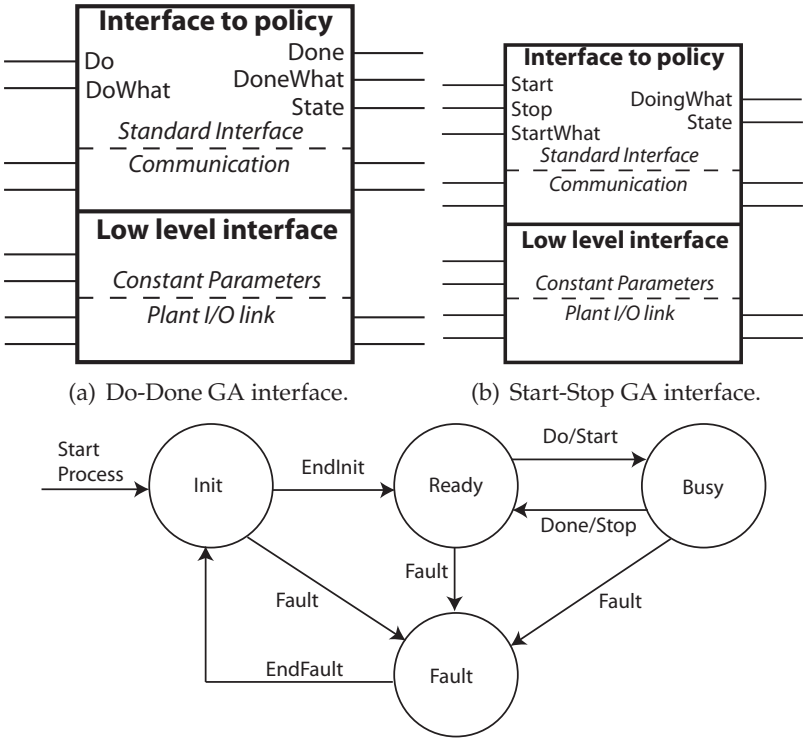
   Constant parameters: contains all the inputs by which it is possible to give a constant value to characteristic parameters of the GA (e.g. in `WorkPieceMotion` GA the input `PositioningDelayTime` by which define the time interval between the activating instant for sensor `WorkPieceSensor` and the instant in which the workpiece reaches the drilliing position.

   Plant I/O link: is the real interface with the plant and contains as inputs all the links to sensors and as outputs the links to actuators. In this way the physical connection between the GA and the plant is completely hidden to the high level control policy.

The GA should then be designed considering as *Reference Model* the event driven evolution in figure 3.7(c). In the depicted automaton it is possible to distinguish the states in the following described.

`Init`: this state is the initial one and becomes active as soon as the GA is activated (usually at the beginning of operations). It represents the state in which initialization actions are performed; the GA moves out from this state when a signal `EndInit` communicates that the initialization operations are concluded forcing the GA to move in `Ready` state.

`Ready`: in this state the GA is ready to perform the desired operation and is waiting for the `Do` or `Start` command to move to `Busy` state.



(a) Do-Done GA interface.

(b) Start-Stop GA interface.

(c) GA evolution model.

Figure 3.7: Interfaces of GAs.

`Busy`: after the command issued by the policy the GA starts performing its required task communicating with the high level policy information on the accomplishment of the function (e.g. information on the quality of the operations). The GA remains into this state until the task is finished and the signal `Done` is raised (Do–Done GA) or until the `Stop` signal (Start–Stop GA) is issued by the policy. In these cases the GA moves back to state `Ready`.

`Fault`: from any state a signal `Fault` (used to communicate some anomalies) can force the GA to move into a `Fault` state in which some counteractions are taken. Note that the `Fault` signal can be both due to external commands (e.g. an alarm issued by an external operator), to internal diagnostics or to wrong logic operations. When the alarm situation is concluded (signal `EndFault`) the GA returns in the `Init` state to be reinitialized.

It is important to stress that each of the states just described represent a set of states; in this sense the automaton in figure 3.7(c) plays the role of a logic design pattern similarly to GEMMA diagram (see [65]).

To conclude this Section we briefly summarize the design procedure based on GAs and introduced in section before:

1. Identify basic actions of the process;
2. Define Do-Done actions;
3. Define Start-Stop actions;
4. Identify the GAs by grouping actions with overlapping sets of sensors or actuators;

5. Design each GA by:

   - Defining its interfaces;
   - Designing the actuation logics according to reference model in figure 3.7(c);
   - Designing the internal diagnostics and quality assessment procedures (not considered in this work);

6. Design the high–level policies

3.4.1 Types of actions

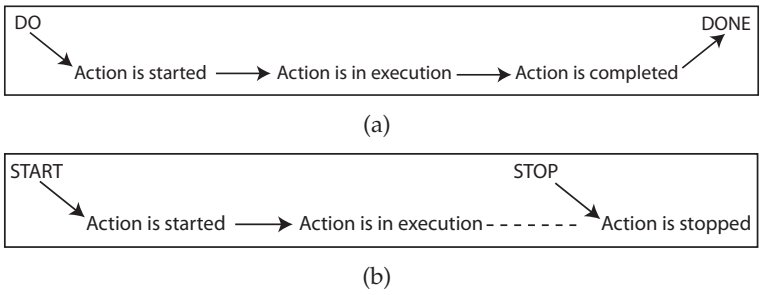


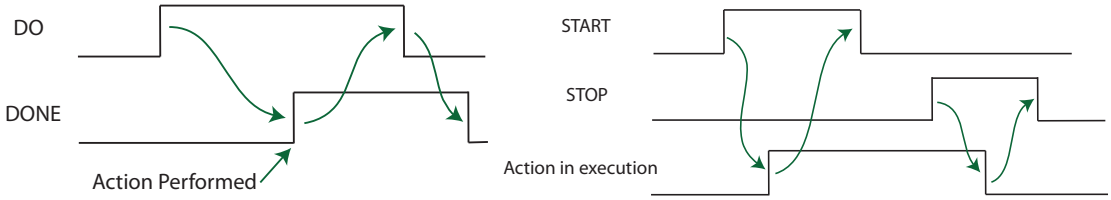
Figure 3.8: Characteristics of Do/Done actions (a) and Start/Stop actions (b)

In section before a procedure to define GA and its characterization was presented, in this section there are some remark on types on GA link on types of actions. Typically, actions performed by an automatic machine can be subdivided in two macro families depending by it's duration: it's possible to recognize *Do/Done* actions and *Start/Stop* actions. In order to clear up the distinction, consider, for example, a system dedicated to the filling of a bottle with a liquid at the temperature of 50°C. Two different type of actions are necessary: actions which structurally terminates after a finite time as filling the bottle until the desired level is reached and actions that could continue for an infinite time and whose termination has to be decided "externally". Referring to the mentioned example, it's possible to analyze the two actions "fill the bottle"until a determinate level and "control the temperature"of the liquid.

- Fill the bottle: when this action is started, the filling of the bottle must continue until the desired level is reached. The action cannot remains active for an infinite time but has a predefined duration decided by the state of the bottle. Actions of this type are called Do/Done actions to emphasize that when the action is required (Do) it remains in execution until its accomplishment (Done).

- Control the temperature: when required, the liquid must have a determinate temperature. The action starts and remains in execution until it is required that the liquid must have the desired temperature; there isn't an intrinsic duration. Actions of this type are called Start/Stop actions to emphasize that when the action is required (Start) it remains in execution until it is no longer required (Stop).

In figure 3.8 is highlighted the difference in duration between the two types of actions. In addition, in the figure 3.9 is depicted the different synchronizations of the two type of actions.



(a) Do-Done GA actions. (b) Start-Stop GA actions.

Figure 3.9: Characteristic of GAs action.

3.5 GA in rapid prototyping

Time-to-market is a crucial issue in developing any industrial product and, consequently, the request of reducing development time in realizing automated industrial plants is getting more and more stringent. Toward this purpose, many tools have been developed to speed-up the testing phase of logic control in the typical development life-cycle. These items are basically *simulation tools* and *rapid prototyping tools*. The latter have been recently introduced to speed up the testing of the final SW or HW/SW implementing the control algorithms, following the software-in-the-loop or the hardware-in-the-loop approaches, respectively (see, beside others, [16], [23] and [63]). In particular, the integration between simulation frameworks and rapid prototyping platforms based on automatic code generation tools has relevantly increased the speed and reliability of the translation from abstract definition of control algorithms to the corresponding implementing software. In this section we'll see how GAs help the designer in rapid prototyping of control logic in complex industrial automated systems.
With the term *rapid prototyping* we refer to a set of methods and tools to solve and verify control problems fast and efficiently resulting in a functioning controller prototype; such techniques include theoretical methods of control engineering from designing a system model and system analysis to the design of a control strategy, test and optimization of the control strategy in a simulation environment, automatic code generation for a real-time system operating a test stand and verification and optimization of the controller with the target object on the test stand. When dealing with complex systems, an efficient strategy is to decompose (usually following functional reasoning) the system and verifying each of these parts, leaving the testing of the integration of the whole system as a final step. The challenge in this task is to choose the right way to decompose the system in order to obtain reasonably simple sub-systems while guaranteeing a certain degree of modularity. To this aim, GA can can play a key role; the idea is to follow the same reasoning used to design the control logic to verify its correctness, dividing the actuation mechanisms from the desired policies while leaning on the concepts used to define GAs in order to have a suitable decomposition of the system. This strategy helps in defining a set of simple and verified actuation *bricks* pursuing the idea of modular, reusable and easily maintainable control software: changing actuation mechanisms or changing coordination policy can be treated separately.
In order to better enlighten the advantages of this technique, we apply the GA concept to a bigger part, than the part presented in section 3.2 ,of FESTO manufacturing system. In figure 3.10 is presented the proccessing station of the system, The station allows three parallel operations thanks to a six position rotary table and three working modules: testing module, drilling module and pushing out device. For an exhaustive description of the processing station the reader is referred to B; in the following the desired sequence of operation for this station is described.

When a workpiece is loaded from the distribution station to the rotary table (signal AVAILABLELOADFORWORKINGSTATION becomes active), the table must rotate to move the workpiece to the first working unit. To this aim the table is actuated through the command signal ROTARYTABLEMOTOR and is equipped with a sensor that indicates when the rotary table is aligned (signal ALIGNMENTROTARYTABLEWITHPOSITIONINGS).

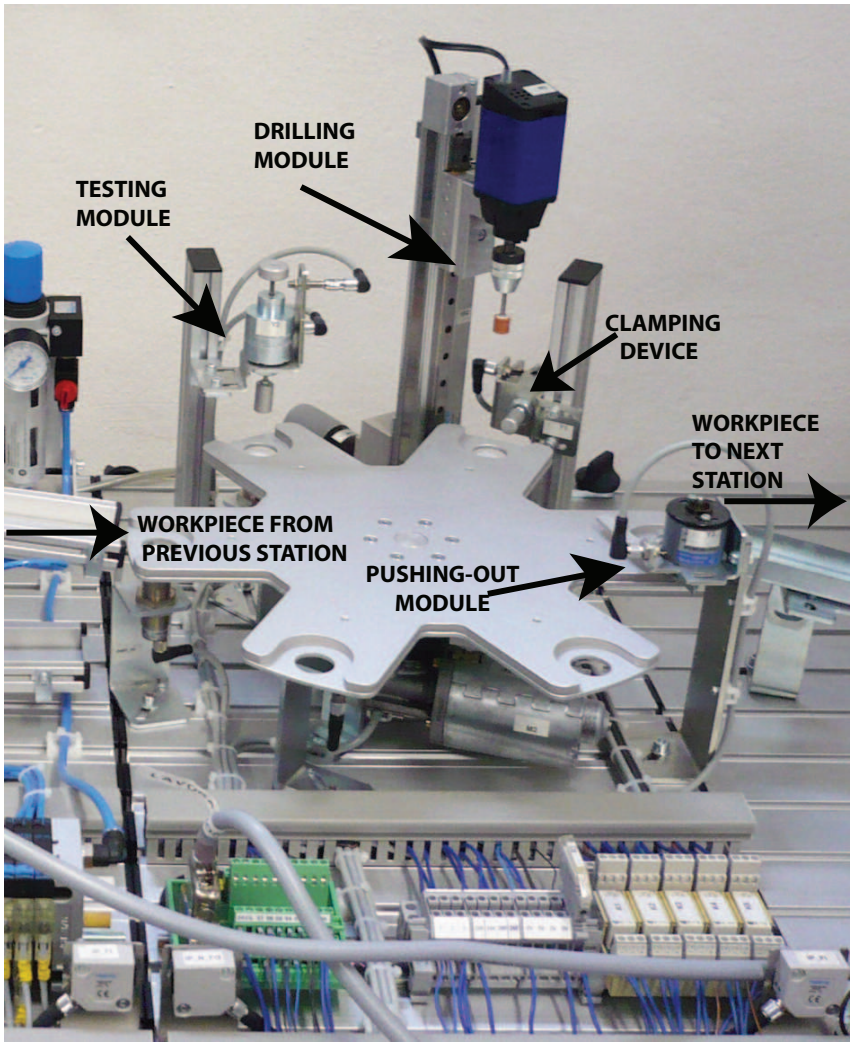


Figure 3.10: Processing station.

When a workpiece is detected in the rotary table, this last makes a 60° rotation (corresponding to the shift of the six working positions) to move the workpiece to the first module of the processing station: the testing module. This module must control if the actual workpiece (its presence is communicated by signal AVAIBLELOADINCONTROLPOSITIONING) has a correct orientation or is upside-down. To do this, the module is equipped by a cylinder actuated through signal TOLOWERCYLINDERTOINSPECTLOAD, which moves downward and checks whether the workpiece is inserted with the opening facing upwards. If the workpiece is in the right position (signal INCONTROLLOADINWRONGPOSITIONTOBEDRILLED is active) it can be drilled, otherwise it must be expelled from the system.

After this testing module, the workpiece is moved to the drilling module by a table rotation;

when the actual piece is under the drilling machine (signal AVAILABLELOADINDRILLINGPOSITIONING is active), the clamping device clamps the workpiece (actuation signal BLOCKINGCYLINDERFORWARDINDRILLINGPOSITIONING) and the drilling operation can start. The drilling machine is activated (actuation signal DRILLINGUNITACTIVE) with a clockwise rotation (actuation signal DRILLINGUNITCLOCKWISE) and is moved downward (actuation signal TOLOWERDRILLINGUNIT) until it reaches its lower limit (signal DRILLINGUNITDOWN). When this position is reached, the driller machine should continue its operation for 2 seconds and, after this time period, it is moved upward (actuation signal TOLIFTDRILLINGUNIT) with a counterclockwise rotation (actuation signal DRILLINGUNITUNCLOCKWISE) until it reaches its higher limit position (signal DRILLINGUNITUP). At this point the drilling operation is concluded and the drilling machine can be stopped deactivating signal DRILLINGUNITACTIVE, while retracting the clamping device (deactivation of signal BLOCKINGCYLINDERFORWARDINDRILLINGPOSITIONING).

With one more $60°$ rotation of the rotating table the workpiece arrives to the last module: the pushing-out module. When a workpiece is within this module (this situation is not recorded by any sensor), a mechanical link actuated by signal EXPELLINGLEVERACTIVE expels it to the assembly station. If the piece was recognized to be upside-down, an alarm light (actuated by signal LIGHTUPSIDEDOWNLOADINEXPELLING) communicates this situation and the system is stopped waiting for a manual removal of the piece. In table B.3, in appendix B, signals used in the testbed are listed as well as their activation meaning.

The first step to define Ga as explained in section 3.3 is to identify basic actions required to the plant; these can be described as follows.

1. *RotaryTableActive*: move the rotary table.
2. *ToLowerCylinderToInspectLoadActive*: move downward the testing device.
3. *ToLiftCylinderToInspectLoadActive*: move upward the testing device.
4. *BlockingCylinderForwardInDrillingPositioningActive*: clamp the workpiece for drilling.
5. *BlockingCylinderBehindInDrillingPositioningActive*: disable the clamp to unlock the workpiece after drilling.
6. *ToLowerDrillingUnitActive*: move downward the drilling device.
7. *ToLiftDrillingUnitActive*: move upward the drillling device.
8. *DrillingUnitClockWiseActive*: rotate the drilling device clockwise.
9. *DrillingUnitUnClockWiseActive*: rotate the drilling device counterclockwise.
10. *ExpellingLeverForwardActive*: move the arm that expels the workpiece.
11. *ExpellingLeverBehindActive*: release the arm that expels the workpiece.

It is important to note that, when defining basic actions, we introduce some redundancy. For example for clamping device, we only command its closure (activation of signal BLOCKINGCYLINDERFORWARDINDRILLINGPOSITIONING), while the release action is implicit in the deactivation of command signal (in this case a spring mechanism allow the release action); however, we list also the clamp unlock action. This is mainly due to the effort of completely decouple

Act.	Sensors	Actuators
1	AlignementRotaryTableWithPositioning	RotaryTableMotor
2	InControlLoadInWrongPositionToBeDrilled AvailableLoadInControlPositioning	ToLowerCylinderToInspectLoad
3		ToLowerCylinderToInspectLoad
4	AvailableLoadInDrillingPositioning	BlockingCylinderForwardInDrillingPositioning
5		BlockingCylinderForwardInDrillingPositioning
6	DrillingUnitDown DrillingUnitUp	ToLowerDrillingUnit ToLiftDrillingUnitUp
7	DrillingUnitDown DrillingUnitUp	ToLowerDrillingUnit ToLiftDrillingUnitUp
8		DrillingUnitClockWise DrillingUnitActive
9		DrillingUnitUnClockWise DrillingUnitActive
10	UpsideDownLoadRemovedInExpelling	ExpellingLeverActive LightUpsideDownLoadInExpelling
11		ExpellingLeverActive

Table 3.2: List of sensors and actuators associated to actions.

policy from mechanisms: the policy is interested in issuing the unlock command, while completely disregarding the physical mechanism that allow the action. Role of the GAs devoted to accomplish such actions will be to map the actions into correct signal sequences.

Once we have defined basic actions, the second step is to associate to each action the set of sensors and actuators that are involved (see Table 3.2.

At this point, following the procedure presented in 3.3, it is possible to define the following GAs[1]:

- **RotaryTable** (DD): implements action 1;
- **ControlUnit** (DD): implements actions 2-3;
- **BlockingCylinderInWorkingStation** (DD): implements actions 4-5;
- **DrillingUnit** (DD): implements actions 6-7;
- **DrillControl** (SS): implements actions 8-9;
- **ExpellingUnit** (DD): implements actions 10-11;

Logic control of the drilling module deserves a further explanation; three GAs are devoted to such task (namely **BlockingCylinderInWorkingStation**, **DrillingUnit** and **DrillingControl**). In fact, the drilling operation issued by the policy can be further decomposed in sub-operations: (i) lock workpiece, (ii) drill the workpiece, (iii) extract the drilling unit and (iv) unlock the workpiece. Moreover *drill the workpiece* action can be decomposed in (a) activate the drilling unit with a clockwise rotation, (b) move downward the drilling unit until its lower limit is reached and (c) wait for 2 seconds in this position while the *extract the drilling unit* action means (a) lifting the drilling unit while (b) rotating counterclockwise. Such sequences of actions reflect in

[1]Label SS stands for Start-Stop GA, while label DD stands for DoDone GA.

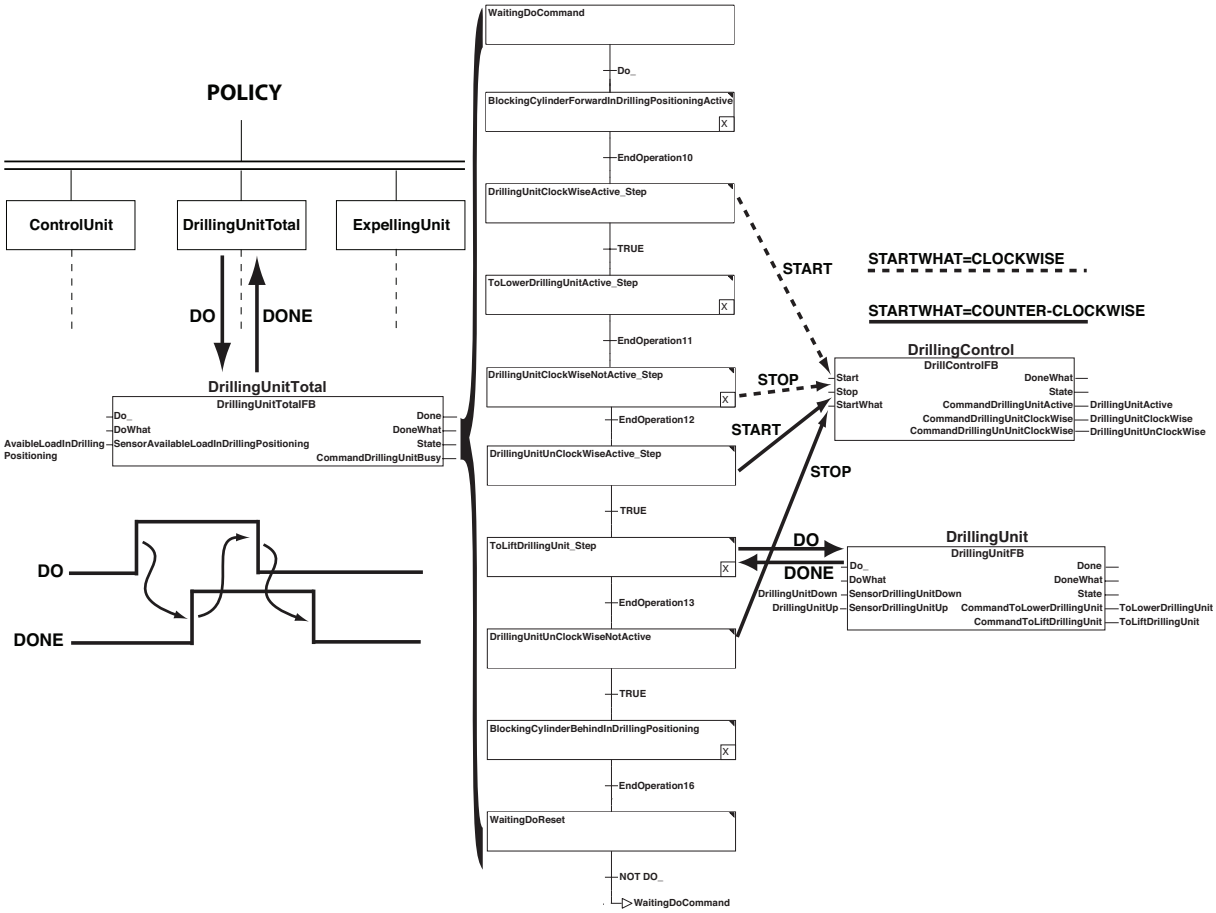


Figure 3.11: Hierarchical GA DrillingUnitTotal.

a hierarchical organization of the involved GAs, as depicted in figure 3.11. More in details an *high level GA* can be introduced (named **DrillingUnitTotal**) which is directly interfaced with the policy; this GA embeds a sub-policy to correctly coordinate low level GAs **BlockingCylinder-InWorkingStation**, **DrillingUnit** and **DrillingControl**. Doing this, the policy is aware just of the existence of a drilling macro-action, the sequence of actions that corresponds to this macro-action is hided in the **DrillingUnitTotal** GA, while physical implementation of each action is considered in low-level GAs. For further details on the implementation of GAs and of the policy for the testbed the reader can see appendix B

The first step of rapid prototyping procedure has been the validation of the simulation model; to this aim each single GA must be tested both in simulation and on real plant to check consistency. Note that, at this stage, just the actuation mechanisms that correspond to basic actions matter; in fact the policy, i.e. the coordination of actions, is useless when testing the model. Because of this fact, at this stage a very simple policy has been considered, corresponding to the workflow of a single workpiece.

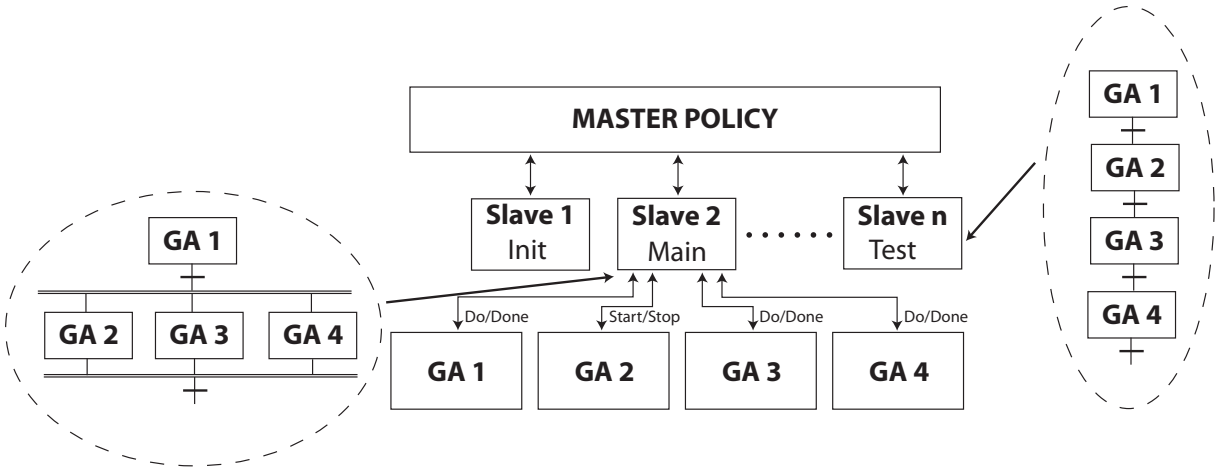


Figure 3.12: An example of policy manager with GA approach.

3.6 Conclusions

In chapter 2 and in section 3.1 it was presented different approach to design control logic in industrial automated systems and their limitation. In this chapter it was present an architecture for industrial automated systems based on the novel concept of the *Generalized Actuator*.
The design procedure to realize automation functions exploiting a clear and structural separation between policies and actions has been introduced To this aim a novel entity denoted as *Generalized Actuator* (GA) is defined in order to (i) encapsulate *actuation mechanisms* separating them from high level control policies in order to hierarchically manage the plant, (ii) support hardware virtualization, component interoperability and reusability and (iii) make easier the design of diagnostics, reconfiguration and quality check functionalities exploiting a distributed and hierarchical approach. Usually in automated systems the machine has different operation mode, like initialization mode, test mode, etc. These different operation modes have different "sub policies"which interact with the same actuators and sensors but with a different coordination, like it's possible to see in figure 3.12. From this point of view is clear as GA approach help the designer in the design of the entire control logic, in fact the different policies use the same GA but only in a different order or coordination, but actuation mechanism defined by GA are the same, we can image a GA like a musician and the policy like the conductor, the musician is always the same but the conductor "drive"the musician in a different way.
Actually the emphases has been put on the improvements obtained in terms of modularity and reusability, but it is also important to stress that the proposed procedure, that reflects into a hierarchical architecture, is the starting step towards a hierarchical diagnostics systems, in which mechanisms fault diagnosis is separated from policy safety verification, and especially towards a hierarchical reconfiguration system in which fault counteractions can be taken separately at mechanisms level and/or at policy level.
In chapter 2 is shown the standard ISA S88 which define a hierarchical approach with the concept of recipe. The main difference in GA approach is the policy is completing separate from the actuation mechanism. In S88 the equipment phase, a sort of low level, implements a part of logic to know if this operation is safe for the process. In this way the low level have to know a part of policy, and this part of policy is dependent from application this means the equipment phase can not be reusable in a different process because the part of logic inside change from ap-

plication to another. Another point is, with the S88 the policy is distributed on all the systems and this cause a difficult to test its correctness.

Chapter 4

The Generalized Device concept

This chapter introduces and analyzes a novel entity called *Generalized Device* (GD) for logic control and diagnosis of field devices in industrial automation. The emphasis has been particularly put on the diagnostic functionalities embedded within the GD entity. In addition, it has been shown that, following an inheritance principle, starting from the basic GD, other typologies can be easily derived. The case study enlightens how to implement GDs for the control of field devices typically used in automated manufacturing systems.

4.1 Actuation mechanism

In chapter 3 it was presented the novel concept of the Generalized Actuator, and it was explained as GA can help the designer to develop the control logic in a modular way. One limitation of GA approach is that that GA has a standard interface but a specific GA component could be reused only if the plant part it handles is reused. In this chapter we'll see how it is possible to define a new layer in architecture independent from hardware of a industrial automated systems, this new entity is defined *Generalized Device* (GD). Analyzing the typical sensorial/actuation equipment in a industrial automated systems it's possible to find different kind of actuators and sensors coming from different fields, so to define an entity independent from hardware implementation we have to answer to this questions: "Do different field devices really need different control logic?"
The term actuator is used for the device which transforms a signal (usually an electrical signal) into some more powerful action which then results in the control of the process, it is possible to characterized the actuator in this fields:

- compressed air (pneumatics);
- hydraulics;
- electric motors.

A sensor is a device that measures a physical quantity and converts it into a signal which can be read by an observer or by an instrument. Sensors are designed and built to address a very different kind of task and can be so different, some sensors are very simple, like position sensors, and there are sensors with a complicate structure, like vision sensor. Usually on a industrial automated system we can find a numerous simple sensors and perhaps a single vision implementation at best. In this section we'll see the main characteristic of sensors, a first classification can be made respect of information generates by the sensor: (i) a discrete binary (on/off) situation, (ii) a continuous value.
An example of discrete sensor it is a sensor used to determinate if a box had been propelled into a limit switch mounted on a movable stop. The information generates from the sensor is if there is the box or it is not. It cannot tell if the box is getting close, or what speed or force the box has when it hits. A continuous sensor can tell you some value that is likely to change quite quickly, for example a pressure sensor can measures the pressure of a gas or a liquid.
A discrete sensor's output is most likely digital, actuating a switch that completes a circuit or disconnects it. Continuous sensor's output can be either digital or analog. It depends on the technology and circuitry being used. Many times it is an analog value that is converted to a digital signal. An example would be the capacitance sensor used for left turn lanes at traffic lights. Many times a sensor wire that connects with the sensor controller is embedded into the pavement. A car with its significant metal mass changes the capacitance field, and the sensor circuit takes the analog signal and determines the presence of a car, or not presence of the car. Then it tells the traffic light processor to take the option of giving a green left turn arrow or not. This sensor has some threshold value for a car, and will often not reach this threshold for a smaller motor cycle. And one will never trigger the sensor with a bicycle. In the traffic light example, the sensor has its own dedicated circuitry or processor, so the traffic light control could be a simple set of relays, if it is an older model. One of the issues to investigate for one's automation machine is where the analog signal gets processed. And even if the main controller is capable of performing such processing, will it be a burden and drag down the overall machine performance. Sensors with built in processors have become quite popular for this reason, as well as having a system that best matches the sensor to the processor. Proper implementation is easier. If a sensor has analog output, it will most likely need to be converted to digital for the controller to process the information. Almost all controllers work on digital signals internally for the decision process. Some controllers have built-in analog inputs where the signal is converted internally, but smaller and less expensive controllers often do not have this capability and need external conversion.
Another sensor type distinction is that of how the sensor determines its value. Some sensors have a direct physical contact other sensors determine the measurement from a distance. In this sense we can divide sensors in (i) contact and (ii) noncontact. Contact sensors have two major concerns, The first is that contact means that there is usually something physically moving, and that moving closes a switch. The second concern is that by contact sensing, the pressure or drag on the item being sensed may change the process. If one is winding a thin film of plastic, and a contact sensor can occasionally poke a hole in the plastic, the information of the sensor will not correctly. In general, contact sensors can be significantly cheaper than the noncontact equivalent. Some noncontact sensors also require significant knowledge of how to implement and adjust them, while a contact sensor's mode of operation is usually obvious.

Pressure sensor : pneumatic line checking device. It is important to know that the compressed air is on and up to system requirements before assuming that air cylinders will function satisfactorily.

Pressure sensor : hydraulic overload. Excessive pressure could blow out the lines and fittings and be potentially dangerous.

Vacuum sensor : suction cup use. Detect whether a box has been successfully picked up by a robot suction cup gripper.

Temperature : hot glue melt. Many packaging machines will not allow for operation if the hot melt glue is too cold, which would have produced poor box sealing.

Weight : scale. A check to see if a package has received the proper amount of product.

Force : strain gage. Used to detect excessive forces on key members during motion. Good for testing machines.

Metal detecting : safety check. Capacitance sensors used to detect metal filings.

Metal position : distance verification. Capacitance sensors used to obtain distance with non-contact. Also known as proximity sensor.

Human presence : motion detection. Microwave transmission and reception can determine motion and/or distances.

Human presence : thermal detection. Infrared detection of humans vs. machine signatures.

Distances : laser range finding. Send a laser beam in the direction and interpret return signal to obtain very accurate measurement.

Distances : ultrasonic detectors. Send out a sound wave and determine distance with reasonable accuracy at less cost than a laser.

Package tracking : radio frequency. Use radio frequency (RF) tags on boxes or products. Can be queried even if on boxes stacked several deep.

Color detection : optical sensor. Use of smarter detectors to interpret reflected light to obtain color information.

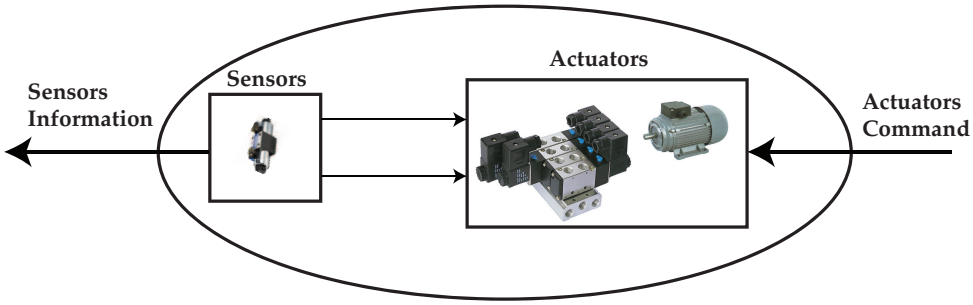


Figure 4.1: Device in industrial automation

This different sensors and actuator can be used to implement an action on a automatic machine. In chapter 3 the procedure to define a GA starts from definition of basic actions. This actions are basic action that manage actuators and sensors link to the actuators to manage the actions. To perform an action an actuator need information from sensors, for example in chapter 3 the

operation of movement of drill unit up and down need information of two position sensors. We can define a device (see fig. 4.1) like a set of actuator and of sensors need to actuator to perform an action. From control logic point of view it is not important to know if the actuator is electric or pneumatic or if the sensor is an inductor sensor or of other kind. The idea of the Generalized Device is based on the concept that different device that perform the same action do not need different control logic, but the control logic will be dependent only by the number of control actions and the number of sensors. This idea will be explained in the next section.

4.2 Devices classification

Taking inspiration from the pneumatic world, it's possible to define the different kind of actuation mechanisms that composes a industrial automated system. Generally, plant is composed of pre-actuators, actuators and sensors. The pre-actuators transform the controller commands into power that is transformed by the actuators into actions performed by the effectors. Sensors are used to inform the controller about the effectors state. Pre-actuators can be classified

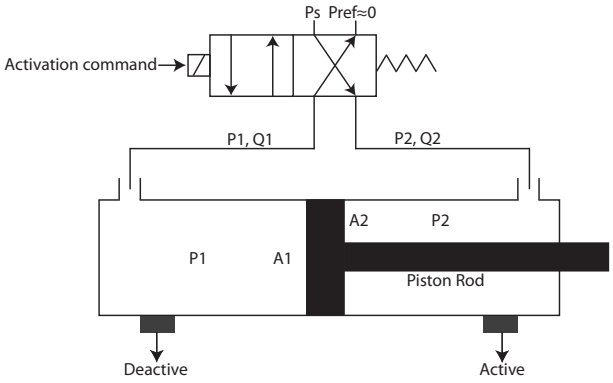


Figure 4.2: Single acting device using a double acting cylinder

in three different types: electrics, pneumatics or hydraulics. Typical electrics pre-actuators are the switch contacts while typical pneumatics or hydraulic pre-actuators are the valves. The valves are the mechanical (or electrical) to fluid interface and the most widely used valve is the sliding one employing spool type construction. Spool valve are classified (see [64]) by *(i)* the number of "ways"flow can enter and leave the valve, *(ii)* the number of lands, and *(iii)* the type of center when the valve spool is in neutral position. A more commonly used classification (the most known) of valve is made by the number of ways and the number of position that the valve takes. Each valve requires a supply, a return and at least one line to the load. Actuators are divided in three different families: electrics actuators (such as motor), pneumatics and hydraulics actuators. The most used pneumatics actuators are pneumatics cylinders; they consist of a piston that moves through a smooth round cylinder or tube. This cylindrical tube must be sealed at both ends with end plates. The end plates are also called end caps or cylinder heads. The volumes between the piston and the cylinder heads are called chambers. The piston is firmly connected to a shaft called a piston rod that exits the cylinder through a hole in one end cap. There are two different type of pneumatic cylinder: single acting and double acting cylinder. In the single acting cylinders, only one chamber can be supplied by the fluid, with a valve is possible to move the cylinder only in one direction. The opposite movement

is performed by a spring. In the double acting cylinders, instead, both the chambers of the cylinder can be supplied and by two valve is possible to move the cylinder in two direction. A sensor is a device that measures a physical quantity and converts it into a signal which can be read by an observer or by an instrument. It's possible to classify the sensors respect to the physical quantity that they measures: for example there are sensors of force, current, position, etc.. Sensors differs also by the type of technology that it uses; there are for example switch sensors, inductive or capacitive sensors etc.. After this brief discussion it's possible to show the different type of actuation mechanisms that is possible to find in a manufacturing systems.

Single acting devices
Single acting devices have only one actuation command that force the actuator to move in a defined direction; the opposite movement can be performed by the actuator itself (think to a single acting cylinder) by, for example, a return spring. The opposite movement can also be performed by the pre-actuator when the actuation command is removed. This second case is depicted in figure figure 4.2. A double acting cylinder is managed by a 4-way/2-position valve electrically actuated by a single command: when the Activation command is raised, the valve move to a position so the left chamber is connected to the pressure supply (*Ps*) while the second chamber is connected to the return (*Pref*) and this cause the extraction of the piston rod. When the Activation command is removed, the valve automatically switch to it's stable position in which the connection of the chambers of the cylinder is opposite respect to the case just described; this cause the retraction of the piston rod. In this way it's possible to obtain a single acting device using a double acting cylinder. To signal to controller about the position of the piston rod, an arbitrary number of sensor can be used: however, typically a maximum of two sensors are used to inform the controller about the completely retracted and completely extracted positions. In the figure one is depicted the case in which are present two different sensors (that furnish signals *Active* and *Deactive*) to indicate both the ends position of the piston rod.

Double acting devices
Double acting devices have two actuation command that force the actuator to move in two direction. In this case, the device can be blocked in any position by simply deactivating both the actuation commands while single acting devices can stay (in a stable way) only in two positions. A double acting device is depicted in figure figure 4.3.

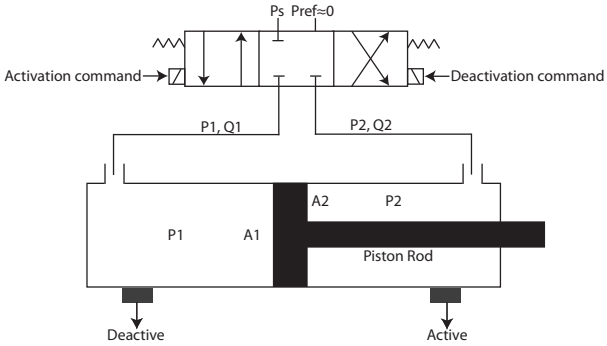


Figure 4.3: Double acting device using a double acting cylinder

A double acting cylinder is managed by a 4-way/3-position valve electrically actuated by

two different commands: when the Activation command is raised, the valve moves to a position so that the left chamber is connected to the pressure supply while the right chamber is connected to the return and this cause the extraction of the piston rod. When the Deactivation command is raised, the valve moves to a position so that the right chamber is connected to the pressure supply while the left chamber is connected to the return and this cause the retraction of the piston rod. When both the actuation command are removed, the valve moves to a position so that both the cylinder chambers are insulated and the piston rod stay in its actual position. Also in this case are present two sensors. It has been mentioned that typically a maximum of two sensor are used to inform about the device state because more than two sensors will introduce complexity without an improvement in terms of performance or safety. Depending by the number of sensors that are present it's possible to give another classification of devices: there are the single feedback devices (if only one sensor is present), double feedback devices (if two sensors are present) and no feedback devices if no sensor is present.
Summarizing all the aspects just presented it is possible to tell that are present the following type of devices:

- Single acting devices (SA)
  - With double feedback (DF)
  - With single feedback (SF)
  - Without feedback (NF)
- Double acting devices (DA)
  - With double feedback (DF)
  - With single feedback (SF)
  - Without feedback (NF)

The different devices topologies are depicted in figure 4.4.

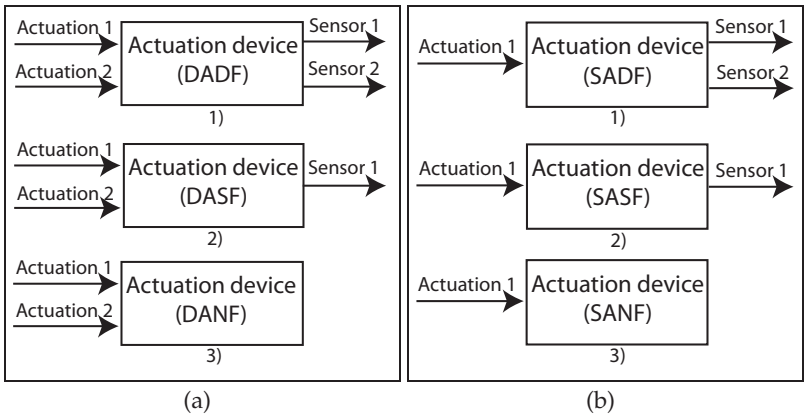


Figure 4.4: Different type of devices: single acting devices (a) and double acting devices (b)

4.3 A hierarchical multi-layer architecture

The role of the control logic in an AMS consists, essentially, on proper management of all field devices so that the overall system behavior fulfils some assigned target requirements. For this

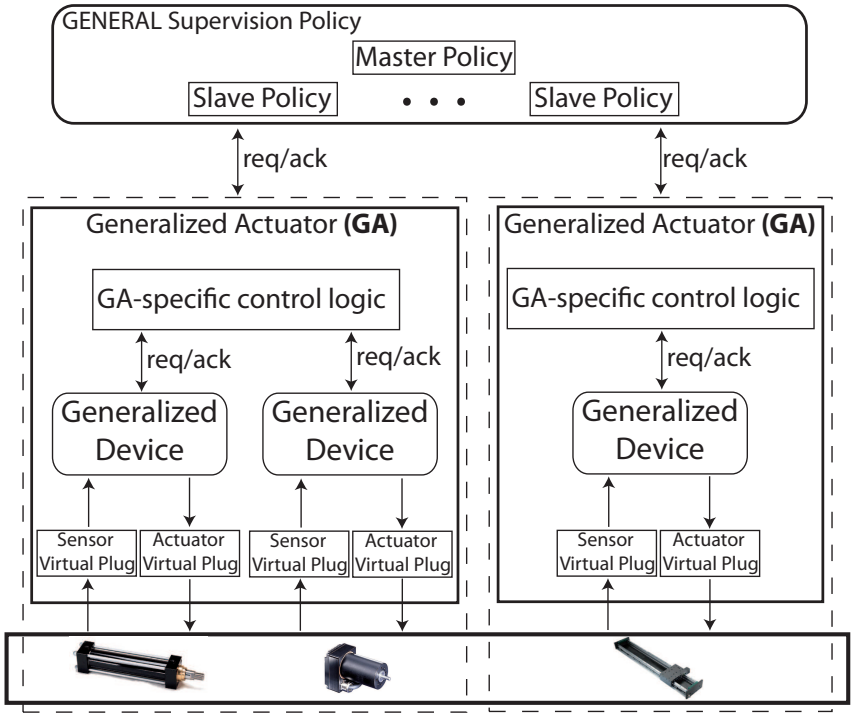


Figure 4.5: The proposed hierarchical multi-layer architecture.

purpose, it has to perform the following three basic tasks: (i) select the system operational mode on the basis of both the commands coming from the user and the controlled plant conditions; (ii) handle the correct sequence of actions involved by the selected operational mode; (iii) drive the field devices to accomplish the desired actions

A schematic overview of the hierarchical multi-layer architecture that we envision to keep cleanly distinct control logic's policies from mechanisms is depicted in figure 4.5. Policies, dealing with "what to do"and "when to do"issues, are handled, at different abstraction levels, within the upper layers of the architecture. Mechanisms, dealing with "how to do"issues, are confined to the lowest layer of the architecture. The new layer comprising the GDs is explicitly enlightened in figure 4.5 to emphasize the substantial difference with respect to the architecture considered in previous chapter, where the functionalities now assigned to GDs were generically encapsulated within the Generalized Actuators (GAs).

The General Supervision Policy (GSP) is devoted to the management of different operational modes and to the realization of the sequences of actions they imply. The GSP may be suitably further divided in a Master Policy (MP) and a set of Slave Policies (SPs). The selection of the desired operational mode (e.g. initialization mode, manual mode, automatic mode, etc.) is performed by the MP, whereas its accomplishment by the delegated SP. The MP could be standardized following an approach similar to GEMMA (see [65], [19]), while the SPs, differently, are strongly application-dependent.

In chapter 3 it was introduced the concept of the Generalized Actuators (GAs) and it was explained that GAs are devoted to the concrete realization of the actions requested by the GSP, directly acting on the plant (or parts of it), but hiding all plant-related features (e.g., involved field devices, typology of actuators and sensors, dynamics, faults diagnosis) to the GSP. Here we just recall that it exhibits a standardized "general behavior"at the GSP interface, giving

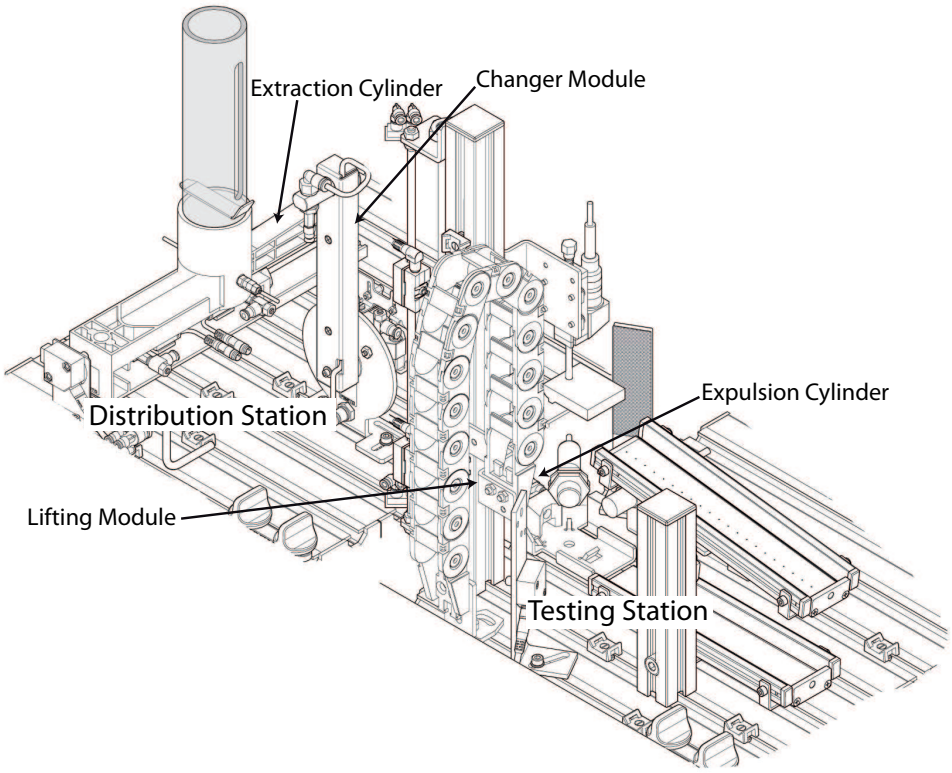


Figure 4.6: The miniaturized AMS used as test bed.

a sort of virtualized image of the specific plant part it handles. Hence, in principle, the architecture, the detailed behavior and the implementation of such entity could be completely application-dependent. This means that the GA interface can be reused, but a specific GA component could be reused only if the plant part it handles is reused. Actually, it is possible to recognize that some basic patterns often recur within GAs to realize actions, particularly when they have to deal with field devices involving only digital control and feedback signals (e.g., single-acting or double-acting pneumatic valves, hydraulic pistons, etc.). Taking the cue from this simple observation, we define GDs as entities abstractly modeling the behavior of the most commonly used digital field devices, regardless their nature, intrinsic features and specific actuation/sensing mechanisms. GDs, as generally-applicable and reusable components implementing basic control/diagnosis schemes, may be profitably embedded within GAs. From this point of view, it is worth noting that, for the sake of generality, a residual GA-specific control logic is associated to each GA in figure 4.5. As a matter of fact, it may be necessary for complex GAs only, to deal with peculiar, yet usually very light, functionalities. As regards the connection of a GD on the plant side, a suitable Sensor/Actuator virtual plug of signals are used. The kind of services that a GD exposes at the interface with the GA-specific control logic, as well as the protocol envisaged for their interaction, will be described in the next section.

To show an example of application of the proposed control architecture of figure 4.5 we consider as testbed a part of a micro flexible manufacturing system produced by FESTO didactic. For a complete explanation of the entire system the reader is refer to see B, in this example only the first two stations (depicted in fig. 4.6) are analyzed. The stations depicted in figure 4.6 are the distribution station and the testing station. The distribution station is composed by a ware-

house that contains up to eight raw bases. One base at a time is expelled from the warehouse by means of an Extraction Cylinder. The unique actuation command of the device causes the extraction of the cylinder (setting the actuation command to 0 cause the retraction of the cylinder); two limit switch sensors indicates the end positions of the cylinder. The extracted base is transferred to the testing station by means of a Changer Module that can be commanded using two different commands (one for each direction). The end positions of the changer module are signalized by two electric microswitch. To the extremity of the module there is a suction cup necessary to hold the raw pieces during the transportation. The purpose of the testing station is to check the colour and the height of a base; if the characteristics of the base are compliant with the user's requirements the base has to be transferred to the downstream station, otherwise the base must be discarded. When a base arrives in the station, a sensor reads its presence. At this point, a colour sensor is used to check the colour of the piece. If the base must be processed, a Lifting Module moves it to an height tester to check if the height is correct. Two actuation commands are used to move the Lifting module while its two end-stroke positions are signaled by two sensors. If all the measurements are correct, the raw piece is moved to the downstream station by means of an Expulsion Cylinder. If the base has not to be processed it is discarded by means of the same expulsion cylinder. One actuation command drives the extraction of the cylinder while, only one sensor, indicates when it is completely retracted. It is possible to classify the different devices that composes the two analyzed stations: the Extraction Cylinder is a SADF device, the Changer Module is a DADF device, the Lifting Module is a DADF device and the Expulsion Cylinder is a SASF device.

The architecture of the hierarchical control software realized for the proposed example is depicted in figure 4.7 it reflects the general one depicted figure 4.5. For the sake of brevity, only the Main slave policy is completely expanded; it is clear that for all the slave policies, the same structure is implemented. For the purpose of this paper we will focus only on the extraction cylinder device. The control software is composed by both the GAs and the GDs entities. The GD is the responsible of the management of the field device and, as above described, it also performs the first level of diagnosis; when a fault is detected, the GA moves into the Alarm state where some counteractions are taken (for example a SADF device, after the detection of a sensor fault, can be reconfigured to work as a SASF device as explained in the previous section). The GA is then the responsible of a first level of reconfiguration.

4.4 The Generalized Devices

In the following will be explore and analyze the SA devices, starting from the more general and widely used SADF type, then extending our considerations to the SASF and SANF types. Figure 4.8 shows the structure of the SADF GD and its interfaces. The interface on the plant side involves the digital output A (Activation) controlling the field device actuator and the two digital inputs a (Activated), d (Deactivated) coming from the field device sensors. On the GA-specific control logic side, the GD exposes a standard interface for requesting/notifying the execution of the two elementary services it provides. More in detail, the GD accepts as commands RA (Activation Request) to activate the device and RD (Deactivation Request) to deactivate the device. Service completion is notified by the GD via AA (Activation Ack) and AD (Deactivation Ack), respectively. The device Activation Time (TA) and Deactivation Time (TD), reflecting respectively the maximum amount of time estimated for its activation and deactivation, as well as the desired initial Activation value, are configurable via the configuration/communication interface. The same interface permits to communicate all detectable device faults.

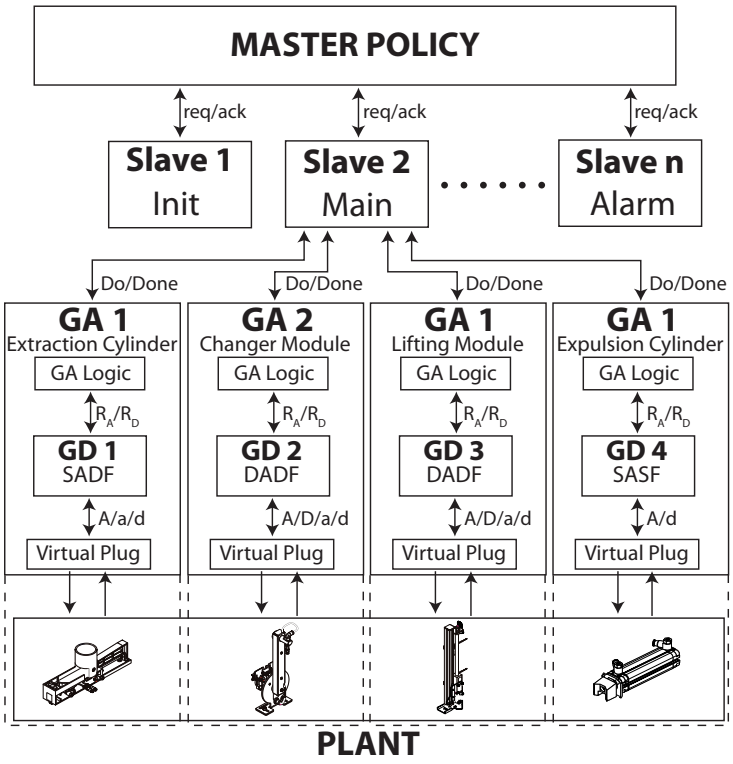
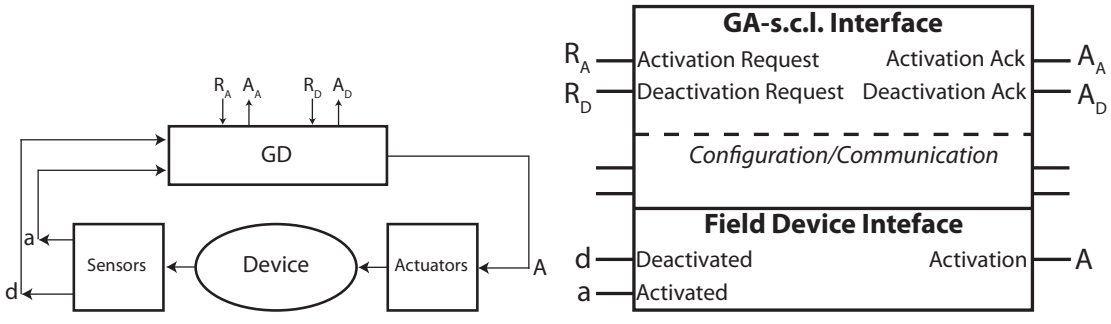


Figure 4.7: The hierarchical multi layer architecture envisaged for the control of the AMS used as test bed.



(a) SADF model.

(b) SADF interface.

Figure 4.8: The SADF GD and its interfaces.

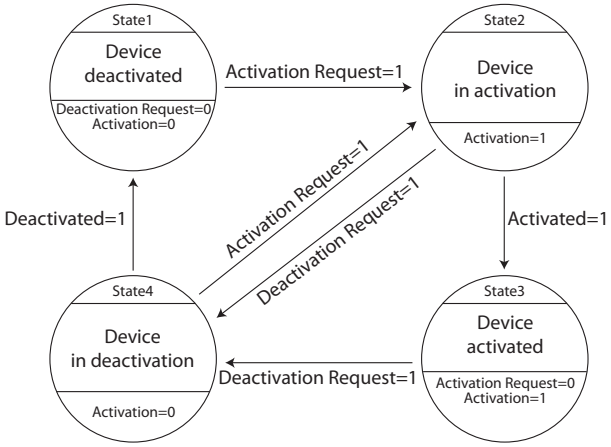


Figure 4.9: FMS modeling the behavior of the SADF GD.

The behavior of the SADF device in nominal operating conditions (i.e., following initialization) is modeled by the FSM depicted figure 4.9. Three relevant aspects have to be pointed out. (i) A positive polarity is assumed for all GD inputs/outputs, meaning that the logical name assigned to each of them unambiguously identifies, in positive logic, the role it plays. (ii) A very simple, yet effective, handshaking protocol is here envisioned to guarantee proper coordination between the GA-specific logic and the GD. The former requests the execution of a service by asserting either RA or RD, the latter notifies service completion by de-asserting the request.[1] (iii) The GA may request the GD to abort the execution of a service by simply asserting the opposite request.
The interface shown in figure 4.8(b) has the following structure:

1. **GA - interface:** this section represents the input/output section between the GD and the GA. It can be further decomposed in two subsections separating the standard interface between the GD and the GA and the interface for the configuration parameter of the GD and its communications with the GA. In this pat GD embeds all command inputs for the GD an the outputs that communicate the actual state of the GD. More in detail, the single acting GD will receive as command the Activation request signal to activate the device and the Deactivation request signal to deactivate the device. Fore a major clearness think to a single acting cylinder: this type of device is normally in a specific position (i.e. retracted) and it will move into the opposite position (i.e. extracted) as soon as the actuation command is raised that is the device is activated by the Activation request. The Deactivation request signal cause the device to return in its initial position. The double acting GD will receive as command the Activation request signal to activate the device, the Deactivation request signal to deactivate the device and the Stop signal to stop the device operations. The Activation request and Deactivation request signal have the same function as for the single acting GD, the difference with single acting GD is the presence of the Stop signal: single acting devices have only two stable position while double acting devices have the possibility to be blocked in any position. When the double acting GD receive the Stop signal it block the field device in the current position. For both devices the unique output of this section is the State signal used to communicate the actual state in which GD is evolving.

[1]Referring to an IEC61131-3 FB implementation of the GD, this is possible defining

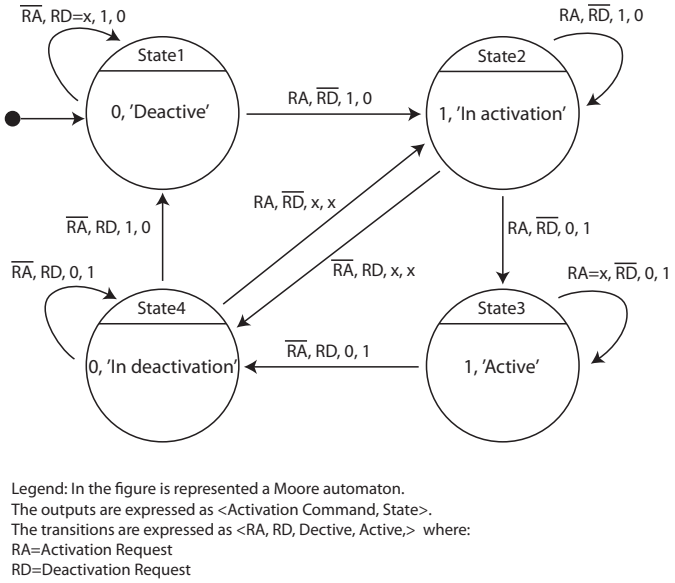


Figure 4.10: State diagram for the single acting GD with double feedback

Configuration/Communications: represents all the signals that can be used to configure some parameters of the GD such as the number of feedback, the diagnostic timers, the initial state etc. and all the signal that can be used by the GD to communicate with the GA such as alarm signals or a sensor state.

2. **Field device interface:** this section contains as inputs all the links to sensors and as outputs the links to actuators of the device that manages. More in detail, for both single and double acting GD, the inputs of this section are the signal Activated and the signal DeActivated which are the link to the sensors that indicates when the device is active or deactivated (e.g. the position of the actuation mechanisms). For the single acting GD the output of this section is the signal Activation command which is the link to the unique actuation command of the single acting actuation mechanisms. When this signal is high the device is activated while when this signal is low the device is deactivated. For double acting GD the outputs of this section are the Activation Request signal and the Deactivation Request signal. With the combination `Activation Request=1, Deactivation Request=0` the device is activated, with the combination *Activation Request=0, Deactivation Request=1* the device is deactivated while with the combination *Activation Request=0, Deactivation Request=0* the device is blocked in its actual position. The combination with both request high is not permitted.

A more detailed structure of the single acting GD is shown in figure figure 4.4. The automaton of figure figure 4.10 is the automaton for the nominal operation of the GD; for simplicity it is supposed that an initialization procedure is performed before starting the nominal operation. This is necessary for bring the device in a known configuration; the nominal automata can then start from the initial state (`State1` of figure) which corresponds to the initial known configuration of the device. In the state `State1` the GD is deActivated with the Activation Request at a low logic level and then can receive only the Activation request command; the Activation request cause the GD to evolve in the state `State2` where the Activation Request is raised.

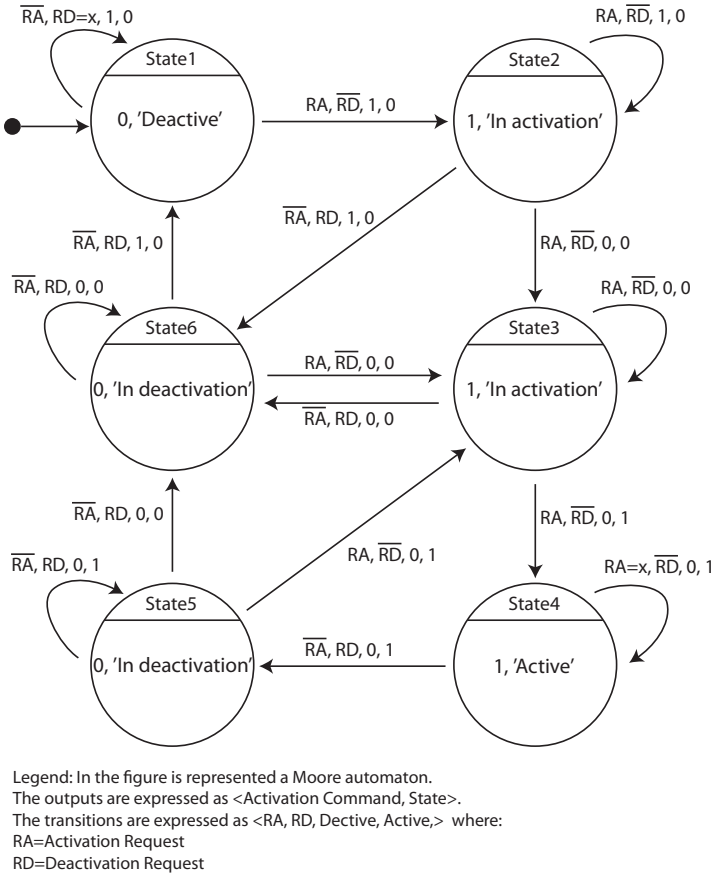


Figure 4.11: State diagram for the single acting GD with double feedback

The device is then in activation; before the completely activation of the device it is expected that the sensors combination changes from Activated=0, Deactivated=1 to Activated=1, Deactivated=0 (`State3`) when active. At this point the device can only be deactivated by rising the Deactivation request signal. If during the activation of the device (state `State2`) the Deactivation request signal is raised (and contemporary the Activation request signal is removed) the GD passes in the deactivation states (`State4`). When the device is in deactivation there is the same behavior.For the control modeling this description is complete but in order to simplify the diagnostic it is necessary to introduce an automaton with 6 states instead of 4: this model is said to be `Diagnostic Oriented` and is depicted in figure figure 4.4. Before the completely activation of the device it is expected that the sensors combination changes from Activated=0, Deactivated=1 to Activated=0, Deactivated=0 (`State3`) and then to Activated=1, Deactivated=0 (`State4`) when active. At this point the device can only be deactivated by rising the Deactivation request signal; also for the deactivation sequence it is expected the same changes in the sensors combination as for the activation sequence (states `State5` and `State6`). It must be taken into account that the device activation (or deactivation) could require a lot of time and this time is then necessary for the fault signal. With the 6 states model the introduction of a complete diagnostic function is simplified respect to the 4 states model. The double acting GD can be implemented through an event-driven automaton which states generally evolve like depicted in figure 4.12. In this case it is directly represented only the Diagnostic

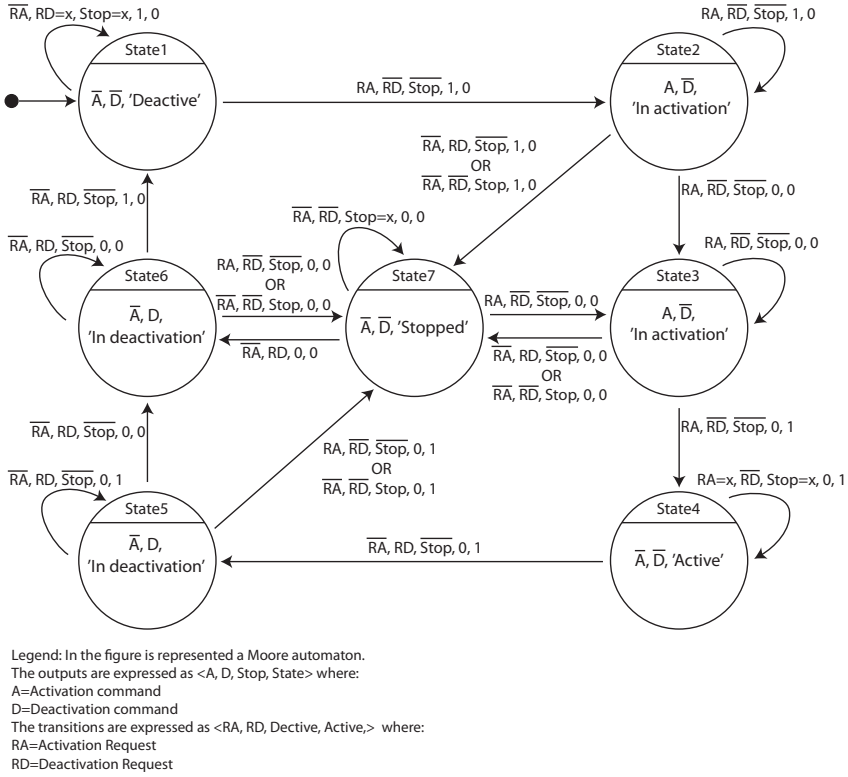
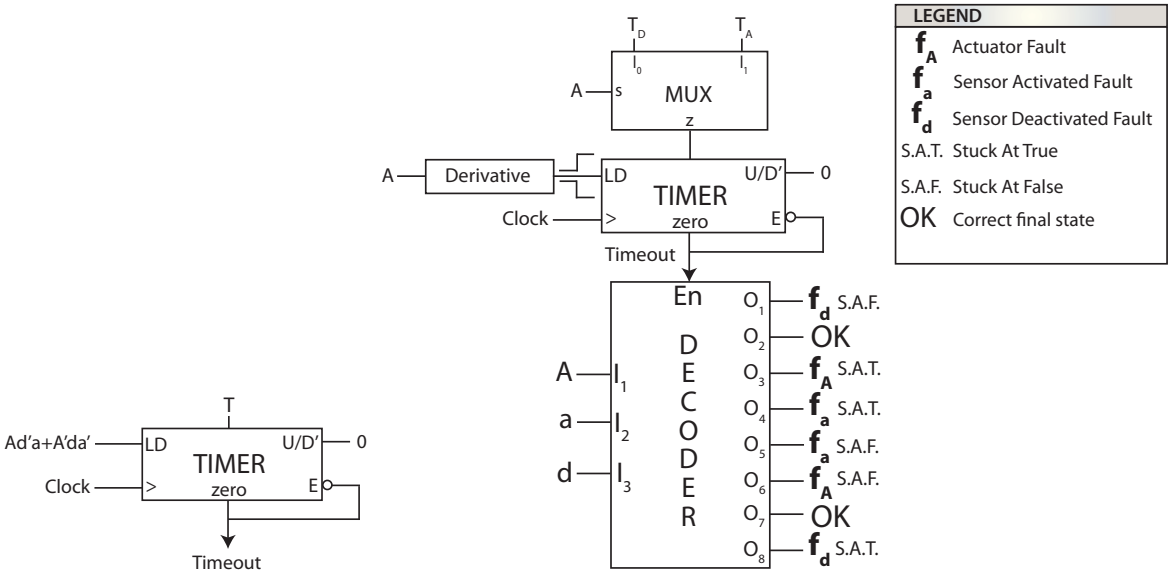


Figure 4.12: State diagram for the double acting GD with double feedback

Oriented model. For the double acting GD is possible to make the same considerations as for the single acting GD but with some modifications. Instead only one actuation command in this case there are both the Activation Request signal and the Deactivation Request signal and in addition to the Activation request and Deactivation request signals there is also the Stop signal. In the figure figure 4.12 is depicted the response of the GD to the different requests. The two automaton are almost identical, the unique practical difference is the presence of the state `State7` in the automaton of double acting GD. The GD is in this state when the actuation mechanisms is stopped between the two active and deactivated positions by rising the Stop signal. In this state the device passes also during the direct changes between activation and deactivation procedures to avoid the possible Activation Request=1, Deactivation Request=1 combination during the transitions.

The two presented automatons represents the GD behavior in the double feedback case: these models remains the same also in the case of single feedback or no feedback simply modifying the conditions of transition. The sensors that are not present must be substituted by a timer which drive the automaton evolution at the same way as the sensor signal. However it is not necessary to design a GD for each feedback typology but only two GD must be designed that are the single acting and the double acting GDs: using the configuration signals introduced early, is possible to configure the GD for the operation as single, double or without feedback.



(a) A simple and common approach to fault diagnosis.

(b) A refined approach to fault diagnosis with activation deactivation timers.

Figure 4.13: Fault diagnosis approach.

4.5 Fault diagnosis functionalities

The presented approach gives the possibility of easily embedding within the GD also diagnostic functionalities. Clearly the diagnosis cannot be effectively performed unless considering the dynamic of the controlled device. A conceptual scheme of a simple and commonly used approach, envisaging a similar time for device activation and deactivation (TA = TD = T), is depicted in figure 4.13(a). A timer is exploited to deal with the device dynamics. When the field device is operating as expected in steady state conditions, that is when the inputs from sensors coherently reflect the condition established by the actuator, the Load input of the timer prevents the assertion of output (Timeout) devoted to signaling a (really generic) faulty behavior of the device. Upon activation or deactivation of the device, the timer starts its countdown. When the timer expires (the zero output becomes true), and the generic fault message is asserted. This approach, completely ignoring the cause-effect relationship existing between A and a, d, has two drawbacks: a faulty behavior of either sensor is reported with a T delay; an unexpected bouncing of either sensor at a frequency less than 1/T is undetectable; unexpected, yet persisting, temporary configurations of either a or d, each lasting less than T, are undetectable. An improvement may be addressed by considering the conceptual scheme of figure 4.13(b). The cause-effect relationship between A (cause) and a, d (effects) must be taken into account. The Load input of the timer is driven by the derivative of A; at any slight variation of A the timer starts its countdown. The decoder has been introduced to allow the detection of the faulty sensor/actuator. When the timer expires (zero output becomes true), the decoder is enabled and the correct diagnostic information is generated. A MUX, driven by A, is used to select the correct time: if A=1 the device is activated and then, the activation timer TA must be selected, otherwise, if A=0, the device is deactivated and the deactivation timer TD must be used. The last proposed method, allow to solve all the drawbacks of the method of figure 4.13(a). It is important to emphasize that the use of timers is exploited also to deal with the problem of image

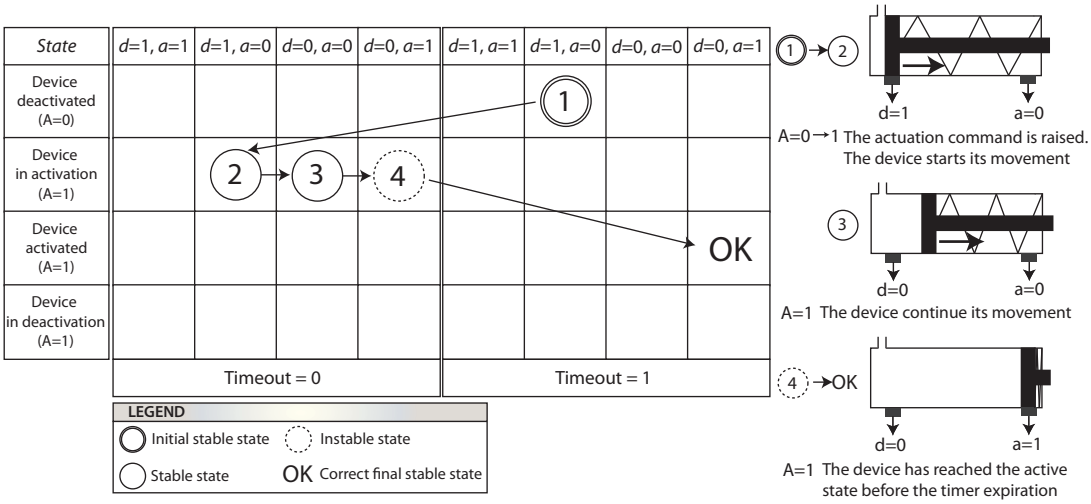


Figure 4.14: The SADF GD state space: state evolution during an activation cycle in absence of faults.

persistence: any decision must be performed just if the sensors configuration is a stable one and not just a spurious transient configuration, an exhaustive discussion of this problem can be find in [55]. Below, we consider a Structured Text (ST) implementation of the approach of figure 4.13(b); obviously, the methodology is independent of the programming language used. Suppose that the Timeout variable is generated as Timeout:=(DeviceTimer=0). The variable DeviceTimer is initialized at the correct value consistently with the value of A and is decreased at regular time intervals (a Clock variable, synchronized with the system time, can be used to this purpose). When the countdown ends, the Timeout variable becomes true.
Note that the diagnosis is not performed for the whole duration of the activation/deactivation cycle. These cycles have a duration approximated by the activation/deactivation times which may be long time intervals. Therefore it may take a long time before a fault is detected. To improve the method, two different timers can be used: the first to monitor the duration of the total cycle whilst, the latter, to check if the device has started the action. The diagnostic function is active if the device is into a stable state (Device activated or Device deactivated) or if the timer expires during an activation/deactivation cycle. To enable the diagnosis when the device reaches a stable state, the DeviceTimer variable will be set to 0 (and the Timeout variable rises to 1) as soon as the device has complete an activation/deactivation cycle. In this way, it is possible to recognize both the so called "static faults" and the "dynamic faults". Static faults are faults that appear when the device is in a stable state, while dynamic faults are faults that appear when the device is in a non-stable state (i.e. Device in activation or Device in deactivation). To explain the dynamic of the fault signals generation we will consider a SADF device that implements the control of a pneumatic piston. In figure 4.14 is depicted the activation cycle of the device without faults. Initially, the device is deactivated i.e., the pneumatic piston is completely retracted. The activation request forces the GD to move to the Device in activation state; the activation command A is raised, DeviceTimer is set to TA and the Timeout variable becomes false. The piston starts moving toward the extracted position. Contemporary, the DeviceTimer is decremented to scan the execution time of the activation cycle. As soon as the piston reaches the active position (a=1), the DeviceTimer variable is set to 0, the Timeout variable becomes true and the diagnosis is enabled. In the situation of figure 4.14 the device

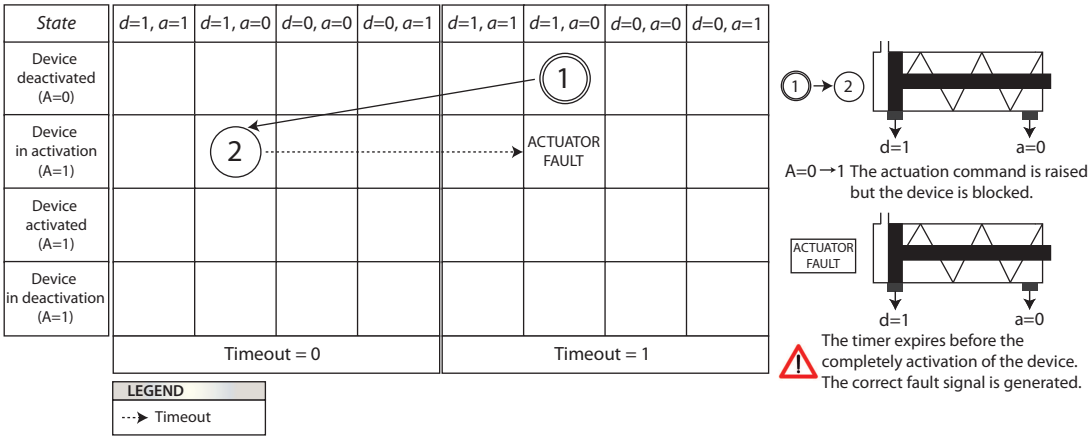


Figure 4.15: Dynamic fault detection in the SADF GD state space.

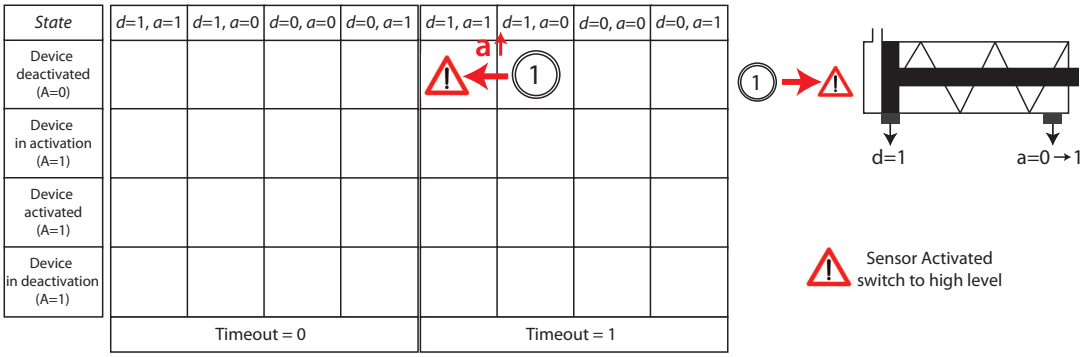


Figure 4.16: Static fault detection in the SADF GD state space.

correctly reaches the extracted position, the sensors coherently reflect the condition established by the actuator and therefore, no faults have affected the device. In figure 4.15 is depicted the situation in which the piston is blocked in the retracted position. At the activation request, A is raised, the DeviceTimer variable is set to TA and Timeout becomes false. The piston, however, is blocked in the retracted position. DeviceTimer is continuously decremented to scan the execution time of the activation cycle. When the DeviceTimer variable becomes 0 (Timeout becomes true) before the activation of the device, the diagnosis is enabled allowing to generate the correct fault signal. The above presented fault belongs to the dynamic fault class. Such as explained before, also static faults must be considered; an example of static fault is depicted in figure 4.16. The pneumatic cylinder is retracted, the GD is in the Device deactivated state and Timeout is true. The diagnosis is therefore enabled. By means of a fault, the a signal becomes (wrongly) true; the correct fault signal is then instantaneously generated. In figure 4.17 are depicted all the diagnostic information that is possible to generate using the proposed procedure, both for dynamic and static faults. It's important to note that the configuration d=1, a=1 both for the Device in activation and the Device in deactivation states is depicted as unreachable state. This is because the evolution toward the activated/deactivated state, is performed as soon as the a/d signal becomes true. Suppose that the piston of figure 4.17 is moving toward the extracted position; suppose also that, during the movement, the sensor d get stuck to the

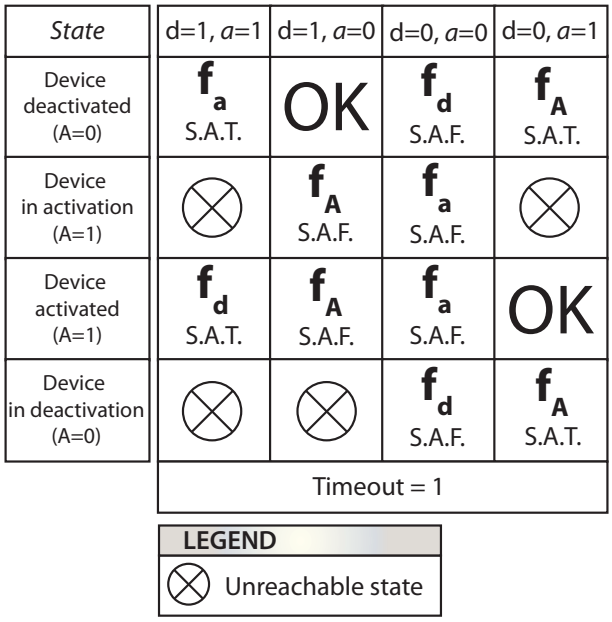


Figure 4.17: Summary of diagnostic signals.

high level. When the piston reaches the extracted position, the sensor a becomes true and the GD instantly evolves in the Device activated state. The diagnosis functionality is enabled and the fault signal is generated. In conclusion, the d=1, a=1 configuration associated to the Device activated and Device deactivated states, is representative for both dynamic and static faults. The diagnostic information is generated with a simple Boolean expression. For example, the reporting of a generic fault of the sensor a, is asserted as follows:

```
FaultSensorDeviceActivated:=((NOT Activation AND Deactivated AND Activated)
OR  (Activation AND NOT Deactivated AND NOT Activated)) AND Timeout;
```

Starting from the analysis just performed, it's possible to easily derive some considerations about the diagnosis for single feedback devices. A single feedback device is equipped with a unique sensor; the missing one is generated by means of a pre-elaboration. For example, if only the d sensor is available, the active device information can be derived as:

```
Activated:=NOT(Deactivated) AND
(DeviceActivated OR DeviceInActivation AND Timeout);
```

while, if only the a sensor is available, the inactive device information can be derived as:

```
Deactivated:=NOT(Activated) AND
 (DeviceDeactivated OR DeviceInDeactivation AND Timeout);
```

It is possible to derive the model of the new device by means of inheritance: the FSM behind the SASF device is the same as figure 4.9 where the lacking information is replaced by the generated one. By a detailed analysis of the presented codes, it's easy to understand that the missing information is calculated as false at the appearance of each fault. Following the idea to inherit also the diagnostic function, it is possible to apply the same reasoning, using the mechanism described in figure 4.13(b) referring to figure 4.17, the diagnostic information that

is possible to give are the same as for the SADF device but, only the columns corresponding to the lacking information identical to false must be considered. More in detail, the proposed strategy allow the detection of the "stuck at false"condition for the available sensor (it does not change its reading from 0 to 1 in a given amount of time when trying to move the device to the stable condition it monitors). For the same reason even the actuator faults cannot be completely isolated: if the available sensor is the active-device sensor it is possible to detect the "stuck active"condition (signal a do not fall to 0 in time), if the available sensor is the inactive-device sensor it is possible to detect only the "stuck inactive"condition (signal d do not fall to 0 in time). It is trivial to prove that for no feedback devices (where both sensors are replaced by a derived information), no diagnosis is possible because of the complete lack of information. Part of the ST code that implements the extraction cylinder GD is the following (in appendix B is reported the entire code):

```
CASE (DeviceState) OF
    DeviceDeactivated:
      DeactivationRequet:=FALSE;
       IF (ActivationRequest) THEN
          Activation:=TRUE;
          DeviceTimer:=ActivationTime;
          DeviceState:=DeviceInActivation;
      END_IF;
    DeviceInActivation:
      IF (DeactivationRequest) THEN
          ActivationRequest:=FALSE;
          Activation:=FALSE;
          DeviceTimer:=DeactivationTime;
          DeviceState:=DeviceInDeactivation;
       ELSIF (Activated) THEN
              ActivationRequest:=FALSE;
              DeviceTimer:=0;
              DeviceState:=DeviceActivated;
       END_IF;
    DeviceActivated:
       ActivationRequest:=FALSE;
       IF (DeactivationRequest) THEN
           Activation:=FALSE;
           DeviceTimer:=DeactivationTime;
           DeviceState:=DeviceInDeactivation;
       END_IF;
    DeviceInDeactivation:
       ......
END_CASE;
```

The ST code just presented simply implements the behavior modeled by the first three states of the FSM of figure 4.9. It is possible to observe the implementations of the handshaking protocol and the management of the DeviceTimer variable to allow the diagnosis of the anomalous conditions.

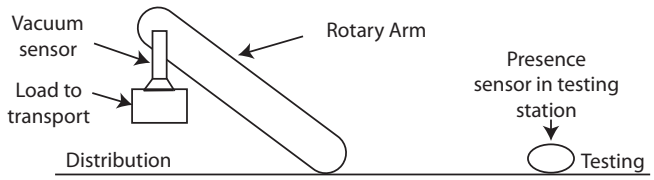


Figure 4.18: An example of high level fault.

4.6 Conclusions

In this chapter has been shown how it is possible to virtualize the hardware using the concept of *Generalized Device*. This concept is based on the idea that the device a further categorization of the devices can be performed on the basis of the number of feedback signals and actuation signal command. The GDs represent therefore the standardization of an actuation mechanism accomplished through the combination of an actuator, one or more sensors and basic control logic. Starting from GDs concepts a hierarchical multilayer architecture for the control of automated manufacturing systems has been presented. The emphasis has been particularly put on the diagnostic functionalities embedded within the GD entity. It has also been shown that, following an inheritance principle, starting from the SADF GD, the other typologies of GDs can be derived by means of simple extensions. The validity of the proposed approach has been experimentally ascertained by a case study derived from the automation field. The case study enlightens how to implement GDs for the control of field devices typically used in automated manufacturing system. The major improvements due to the introduction of the GD components are twofold: (i) standardize logic control policies for basic field devices; (ii) introduce standard diagnostics functions for basic faults in field. Using this architecture we can divide the fault in two level: low level fault and high level fault. A *low level fault* are faults on sensors or actuator and can be directly detects from the GD component. An *high level fault* are fault where two or more device are involving. An example of high level fault is the following: during the transport of a base from one station to the other, the load could fall down from the rotary arm or the load can be efficaciously transported. In testing station there is a presence sensor, so when the load should arrives in testing station and presence sensor is FALSE it is not possible to know if a fault on sensor presence occurs or the load fall down during transportation. The situation is depicted in figure 4.18. The rotary arm is equipped with a vacuum sensor that is been activated when a base is attached to the arm. Therefore, if the base is efficaciously transported, both the vacuum and the presence sensors must be at high logic level. A different configuration of these sensors is an indication of fault. To detect this kind of fault it is necessary use information from different GD, but in this ways there is an architecture also for faults diagnosis.

Chapter 5

A discrete event approach to fault diagnosis in automated system

An important target for control systems, is the correct fault detection and diagnosis and fault tolerant control. For this reason the concepts and techniques of discrete event system theory are receiving increasing attention in industrial automation for control logic synthesis. Specifically, techniques from supervisory control, fault diagnosis, and fault-tolerant control of discrete event systems can be employed to study properties such as controllability, observability, and diagnosability, and to assist in the synthesis of the monitoring and control logic. This chapter presents a general and versatile approach for building structured formal models of complex automated systems in order to facilitate their control and diagnosis, and it takes in consideration active fault tolerant control using techniques from discrete-event system theory.

5.1 Formal verification in industrial automation

Verification and validation (V&V)is the process of checking that a product, service, or system meets specifications and that it fulfills its intended purpose. Verification is a quality control process that is used to evaluate whether or not a product, service, or system complies with regulations, specifications, or conditions imposed at the start of a development phase. Verification can be in development, scale-up, or production. This is often an internal process. Validation is quality assurance process of establishing evidence that provides a high degree of assurance that a product, service, or system accomplishes its intended requirements. This often involves acceptance of fitness for purpose with end users and other product stakeholders.
In software engineering, software testing, and software engineering, Verification and Validation is the process of checking that a software system meets specifications and that it fulfils its intended purpose. It is normally part of the software testing process of a project. With regard to the application of software engineering methods to control problems, an important role is

played by those methods that have as the major goal the construction of correct and reliable systems, by means of analytical and mathematical-based languages, reasoning techniques and formal tools for the specification and verification of software systems. This notations and techniques are called *formal methods*, since both their syntax and semantics are supported by precise definitions, in contrast to those methods that puts more emphasis on the syntactical aspects of the description language, but define the semantics in an informal way. This notations and techniques are called *semi-formal method*. Formal methods represents therefore a way to eliminate ambiguities in the description of a computational system, like a control logic in an industrial automated system. and its desired properties, by means of a *formal specification* and then to apply reasoning procedure to reveal inconsistencies, incorrect behavior and flaws in the specification of the system by means of formal verification procedures.
The formal specification of an industrial system include the description of behavioral properties, but also non-behavioral aspects like requirements real-time constraints or architectural design. Specification methods that put particular emphasis on behavior are, for example, those based on tools and notations derived from DES theory, like automata, Petri Nets and Statecharts. Statecharts is a notation develop to describe state diagram to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction (see [41] and [42]). In fact, the main field for the application of formal methods is that of reactive systems, where safety and real-time constraints are the major requirements, and the most proper way to specify the behavioral properties of a reactive system is in terms of its computational states and its response to events. Other formal methods, adopting a wide variety of notations and mathematical frameworks (i.e. sets, relations, functions, etc.) are: *Z* ([84]), *VDM* ([53]), *Temporal Logic* ([75]).
Model checking is a method to verify that a desired property holds for a finite state model of a systems performing an exhaustive states space search, which is guaranteed to terminate since the space is finite. Of course, the major problem in model checking is to define algorithms and data structures that can manage efficiently large state spaces. In the just case, properties to check are expressed in a certain kind of temporal logic (Linear Temporal Logic or LTL, Computation Tree Logic or CTL) while the system is modeled as a finite state transition system. The search procedure then checks if the given state transition system is a model for the specification. Modal and temporal logics, which are formalisms to state requirements or safety relevant properties of systems, although they are usually not considered as specification languages. Most of the presented languages are designed less with the authoring or communication aspect, and more with the algorithm or tool aspect in mind: concrete syntax for temporal logic is often simplistic, tailored to some particular tool (like a model checker), and lacks the high concentration of syntactic sugar found in specification languages. Because there is a huge number of temporal logic dialects, we only present the main families and common principles. This has the practical advantage that many properties (like satisfiability or the model checking problem) are decidable and can be checked automatically by tools. At the same time, temporal logics are often sufficient to express many interesting requirements that can be imposed on a system. The two main applications of temporal logics in software or system development are the generation of runtime assertions (in this context often called runtime monitors) and static verification by means of model checking. The main characteristic of both modal and temporal logics are the built-in notions of states (or worlds) and transitions between states. The decision whether a modal formula is true is always made in the context of a particular active state. Temporal languages provide larger varieties of modal operators, for instance to talk about states reachable in more than one step from the current state.

Linear Temporal Logic (LTL, see [75])has been proposed in the late 1970s for the verification of computer programs. It treats the "future" as a linear computation path consisting of discrete states. This means that propositions in LTL are always interpreted over individual paths. In case of non-determinism, in which the space of possible program executions has the form of a tree rather than a single linear path, an LTL formula is considered true if and only if it holds on all possible execution paths of the program: paths are implicitly universally quantified. This is the main difference between LTL and the alternative Computational Tree Logic, in which explicit universal and existential path quantifiers exist.
Computational Tree Logic (CTL, see [75]) and CTL model checking were invented in the early 1980s (see). CTL is a branching-time logic and considers the "future" of program executions as a tree (in contrast to LTL), thus making non-determinism directly observable by formulae. Such computation trees can be derived, e.g., from automata or Kripke structures by designating some state as the initial state and unfolding the structure into a (possibly infinite) tree with the designated state at the root. This tree shows all of possible executions starting from the initial state.

5.2 A DES approach for formal verification

Designing control logic that provably satisfies given specifications is a problem of formal verification. Industrial automated systems are very complex systems so for these systems is a challenging task. In chapter 3, chapter 4 and recent work in industrial automation has focused on the concepts of modularity and reusability of the control logic; see, e.g., [11, 27, 94]. To achieve all these objectives, the concepts and techniques of discrete-event system theory are receiving increasing attention in industrial automation for control logic synthesis. Specifically, techniques from supervisory control, fault diagnosis, and fault-tolerant control of discrete event systems (see [17], [79], [97] and [70]) can be employed to study properties such as controllability, observability, and diagnosability, and to assist in the synthesis of the monitoring and control logic; see, e.g., [43, 76, 62]. One of the earliest and most useful methods is *modular control* (see [21]), this method involves designing multiple supervisors as opposed to centralized supervisor. Each supervisor implementing a portion of he control specification. All of these techniques however presume the availability of a complete formal model of the system; building such a model is a difficult task due to its dependence on deep knowledge of the system components and their physical coupling. To tackle the complexity of modeling the overall system in a monolithic manner, researchers have considered decentralized approaches [81] or decomposition methods [29]. In [74] and [81] is proposed a model to fault diagnosis on manufacturing systems. These works exploit the modularity of the model to avoid state explosion, nevertheless the complexity is avoided also using a fault-free-model. On the contrary the modularity should be exploited to deal with simpler models in which it is possible to define post-fault behavior so that the precision of the FDI is improved and the strategy to counteract the faults or to safely react can be formally defined. In the next sections will show a methodology to develop a general and versatile approach for building structured formal models of complex automated systems in order to facilitate their control and diagnosis using techniques from discrete-event system theory. For this purpose, we present a methodology for building in a modular manner the complete model of a complex automated system starting from individual components and their physical coupling. We employ finite-state automata as our modeling formalism. We propose a hierarchical decomposition that separates the control logic into high-level control actions (i.e., what to do) and low-level control actions (i.e., how to do it), coupled

through an interface. This hierarchical decomposition enables reusability of generic models for low-level components. The low-level models incorporate low-level control actions as well as fault detection logic for component faults. This fault detection logic requires modeling the faulty behavior of the components in addition to their fault-free behavior.

The focus of this work is on model-building at the lower level of the proposed hierarchy. We start from generic fault-free models of low-level components such as actuators and sensors. In order to capture physical constraints among these low-level components in a given automated system, we propose the notion of *physical constraint automaton*, which is then coupled with the generic component automata by parallel composition. This approach achieves the desired characteristics of modularity, composability, and reusability. We show how faulty behavior can be gradually incorporated into the model in a modular manner by enhancing the generic component models to include faults and by adjusting the associated physical constraint automata in a manner that captures the effects of these faults on the physical coupling. As a consequence, the entire faulty behavior is obtained by parallel composition of the individual faulty models.

5.2.1 Architecture for supervisory control in industrial automation

In complex automated systems such as manufacturing systems, it is imperative to design the supervisory controller and hence the control logic to achieve modularity and reusability. As discussed in chapter 3 and chapter 4 a crucial point in this regard is the separation of actuation mechanisms from control policies, in order to hierarchically manage the plant. In other words, control should be viewed as the composition of: (i) a set of basic actions; and (ii) a set of coordination policies for the execution of these actions. This is the approach adopted and further elaborated in chapter 3. Designing the control logic using a hierarchical strategy supports component interoperability and reusability and facilitates diagnostic and reconfiguration. Our proposed hierarchical architecture is based on this approach and it is depicted in figure 5.1. This architecture consists of three levels:

- **High level**: This level embeds the control policy of the system; this policy is structured as sequences of activation and deactivation commands for the lower levels. Note that there are no direct actuation commands and that only coordination sensors can appear as physical signals.
- **Interface**: This level contains all the information necessary to let the high level communicate with the low level.
- **Low level**: This level contains the basic control loops for single devices which implement the actuation mechanisms.

A related hierarchical architecture is proposed in [58] [56] [57], where the focus is on the modular verification of safety and nonblocking properties using local components and their interface, in the context of supervisory control problems. In this architecture (called Hierarchical Interface-based Supervisory Control) a system is decomposed into one high level subsystem and multiple low level subsystem which communicate between a well-defined interface. If each subsystems and its interfaces satisfies certain local condition, then the global controllable and nonblocking properties can be guaranteed. The focus of this work is on the modeling methodology rather than on the efficient verification of safety and nonblocking properties. We aim for generic low-level component models that are reusable, leaving to the high level model application-dependent issues regarding “what to do” and “ when to do it”. Actuation

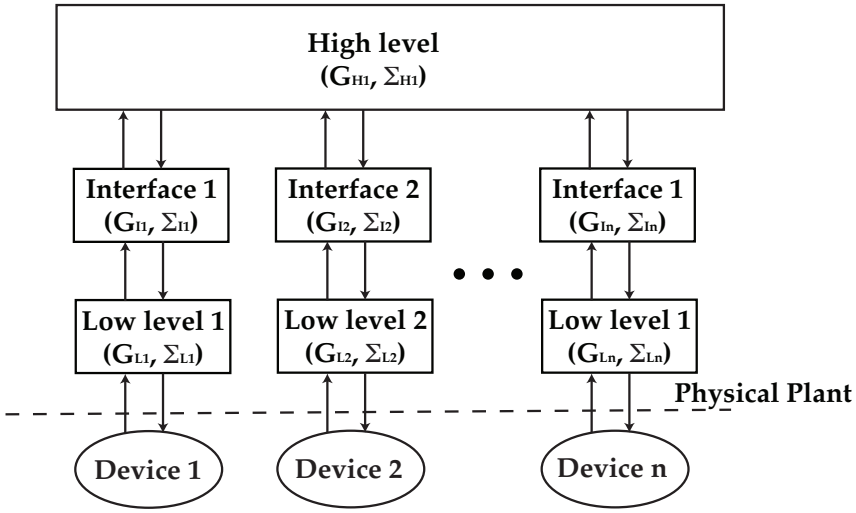


Figure 5.1: Hierarchical architecture.

mechanisms used to implement the desired high-level actions deal with "how to do it" issues, which are dependent on the low-level physical constraints. In this approach, the models of the low-level components are application-independent and thereby reusable. Analyzing common classes of sensors and actuators that equip an automated manufacturing system, we can define general categories of low level devices on the basis of the number of actuation mechanisms, e.g., single or double acting, and on the basis of the number of feedback signals, e.g., double, single, or no feedback; this characterization is presented in chapter 4. To ensure reusability of the low level components, we require that they have a a standard structure and also a standard set of input/output commands. This is where the interface, the middle level of the architecture, comes in. Its role is to map the high level commands into the low level commands. All the components of the architecture can be modeled as finite-state automata (indicated by the symbol G in figure 5.1) over given event sets (indicated by the symbol Σ in Fig. 5.1). In order to ensure the desired properties of modularity and reusability for the architecture, the following assumptions are made:

$$\Sigma_{H1} \cap \Sigma_{Ln} = \emptyset; \tag{5.1}$$
$$\Sigma_{H1} \cap \Sigma_{In} \neq \emptyset; \tag{5.2}$$
$$\Sigma_{In} \cap \Sigma_{Ln} \neq \emptyset; \tag{5.3}$$
$$\Sigma_{Li} \cap \Sigma_{Lj} = \emptyset; \tag{5.4}$$
$$G_{Li} \sim G_{Lj}. \tag{5.5}$$

Equations (5.1), (5.2) and (5.3) ensure the separation between the control policy and the actuation mechanisms, while equation (5.4) guarantees the modularity and reusability of the low level control/diagnosis logic. Controlled devices are modeled as isomorphic automata (with symbol $\sim$ in 5.5 we indicate same structure) and hence their control and monitoring software can be reused.

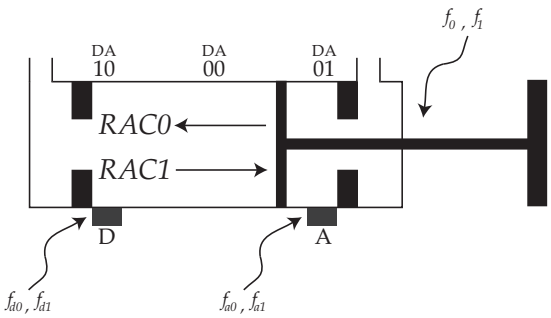


Figure 5.2: Illustrative example: Single acting device.

5.2.2 Model building methodology

To describe the architecture of figure 5.1 we start from the low level. This level associates several low levels and their interfaces with the high level, according to the physical devices comprising the system. In the next section will describe an approach to building automata models for a generic low-level device. The desired model is the composition of automata models of hardware components (actuator and sensors) and models of logic components (control logic, monitoring logic, and logic constraints as timers). The objective of this approach is to embed in the low level not only the low-level supervisory logic, but also the diagnostic logic in order to (i) achieve reusable software as the low level devices are application-independent; and (ii) send to the high level the smallest amount of diagnostic information possible as explained in chapter 4. Starting form a single acting cylinder example shown in figure 5.2 it will clarify these crucial. This device has two end-of-stroke: sensors: sensor A signals when the device is in the activated position and sensor D signals when the device is in the deactivated position. Initially, the device is deactivated, that is, the pneumatic piston is completely retracted. The device is driven by the actuation command AC. When AC is observed to be high, the device moves from left to right unless it was already in the rightmost position (activated). Likewise, when AC is observed to be low the device moves from right to left, again unless it was already at the leftmost position (deactivated). Even this simple device can be affected by six different fault scenarios, each sensor and the actuator can be stuck high, symbols $fa1$, $fd1$, $f1$, or stuck low, symbols $fa0$, $fd0$, $f0$, respectively for sensor A, sensor D, and the actuator. On the device can be occur multiple faults, but on each sensors or actuator only one fault at time, this mean we can have 8 possible combinations of three faults in the worst case: {$fa0$, $fd0$, $f0$}, {$fa1$, $fd0$, $f0$}, {$fa0$, $fd1$, $f0$}, {$fa0$, $fd0$, $f1$}, {$fa1$, $fd1$, $f0$}, {$fa1$, $fd0$, $f1$}, {$fa1$, $fd1$, $f1$}, {$fa0$, $fd1$, $f1$}. When we consider fault on sensor or actuator we consider a physical fault on the component, or a fault on the wire connection between sensor and hardware control logic like PLC or microcontroller, because from control logic point of view it's not possible to distinguish if a fault occurs on the component or if a fault occurs on line connection, for example if on a sensor there is an hardware fault to stuck low the logic out of the sensor, or there is a line disconnection of sensor out on the control logic there is the same effect, it is not possible to distinguish the cause of the fault; With the notation $fa0$, for example, we do not distinguish if fault involves sensor component or sensor connection line to control logic.

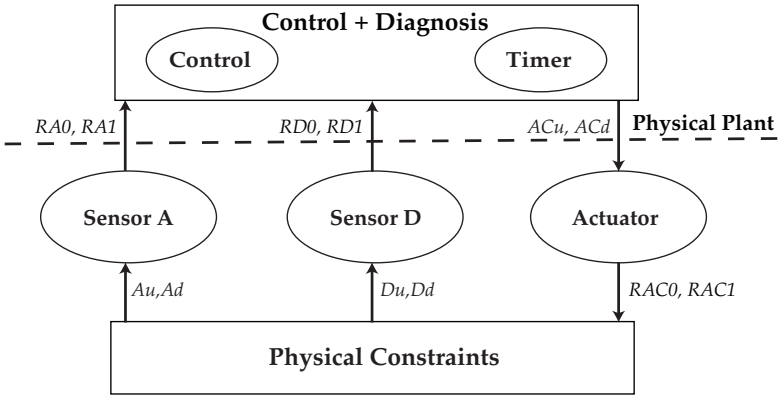
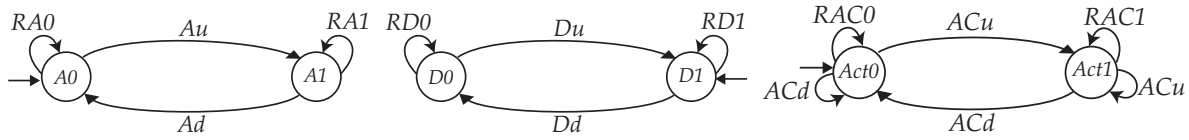


Figure 5.3: Physical and logic components of the low level of the architecture.

5.2.3 Low level

Starting from a single acting cylinder example shown in figure 5.2 it will see the methodology for building the model in nominal condition and fault condition. Low-level component models can be constructed by the interconnection of physical components (actuators and sensors) and logical components (control logic, diagnostic logic and timing logic) as shown in figure 5.3. We start constructing a low-level component by first introducing nominal models for the physical components as shown in figure 5.4. Sensor A has two steady states $A0$ and $A1$; in these states we can have two observable events $RA0$ and $RA1$ indicate if the sensor is read to be low or high respectively, while unobservable events Au and Ad indicate that the sensor output changed from low to high, or high to low, respectively. Sensor D is modeled in an analogous manner (we can define the same events only change label A with label D). The actuator also has two steady states $Act0$ (retracted) and $Act1$ (extended); unobservable events $RAC0$ and $RAC1$ correspond to the movement of the device in one direction or the other. From the point of view of the controller these events are unobservable because the controller cannot observe the movement of the device, it can only read sensor states. Events ACu and ACd are the commands used by the control logic to switch the position of the actuator and hence are observable and controllable, these events can be associate to command send from control logic, for example command `Act:=1` and `Act:=0`

If the beginning the actuator is in state $Act0$ to move the actuator the control logic send the



(a) No fault model $G_{L,Anf}$. (b) No fault model $G_{L,Dnf}$. (c) No fault model $G_{L,Actnf}$.

Figure 5.4: Nominal models of the sensors and actuator.

command `Act:=1` that generates event ACu and actuator will evolve in state $Act1$ and from this state control logic can send a command `Act:=0` that generates event ACd to move actuator in the opposite direction. It is possible to not have the alternate sequence of command `Act:=0`, `Act:=1`, this because can happens that after control logic send command `Act:=1`

in consequence to another task, control logic send again command `Act:=1`, for example after a fault, a reconfiguration control strategy can activate or deactivate all actuators. In this way we can have sequences of commands `Act:=1` or sequences of a commands `Act:=0`, to model this sequences of command we insert self loop on states $Act0$ and $Act1$. From point of view of actuator if control logic send two times the same command, there are no problems because the movement of actuator is always the same. The self loop in each state of actuator model means in each states, we can read information about the last command send from control logic, is like control logic can read the last command it sent.
It is important to remark that the observable events are control commands ACu, ACd and sensors readings $RA0$, $RA1$, $RD0$, $RD1$, while the movement of the device (modeled by events $RAC0$ and $RAC1$) and the rising and falling edges of the sensor readings (events Au, Ad, Du, Dd) are not observable, we can summarize this with the notation of architecture of figure 5.1 in table 5.1: Furthermore, each of the observable events are controllable and each of the unobserv-

SENSOR A	$\Sigma_{A,o} = \{RA0, RA1\}$	$\Sigma_{A,uo} = \{Au, Ad, fa0, fa1\}$
SENSOR D	$\Sigma_{D,o} = \{RD0, RD1\}$	$\Sigma_{D,uo} = \{Du, Dd, fd0, fd1\}$
ACTUATOR	$\Sigma_{Act,o} = \{ACu, ACd\}$	$\Sigma_{Act,uo} = \{RAC0, RAC1, f0, f1\}$

Table 5.1: Observables and unobservables components events.

able events are uncontrollable. The controllable sensor events indicate that the time at which a sensor is read can be controlled.
The models in figure 5.4 consider the actuator and sensors as acting alone, when they are interconnected into the device as in figure 5.2, they interact following a set of physical constraints based on the physical structure of the device (a similar idea is presented in [89]). The physical constraints are the interaction between components and it derives from physical structure of the device. For example after a control command to actuator device starts to move, in a no fault situation the sensors value change with a well define sequence depending from the mechanical connection of the components. To have a complete model of the device we have to model also this physical constraint, it can be modeled by analyzing the behavior of the device shown in figure 5.5. When the device is in the deactivated position sensor D and sensor A read 10. After

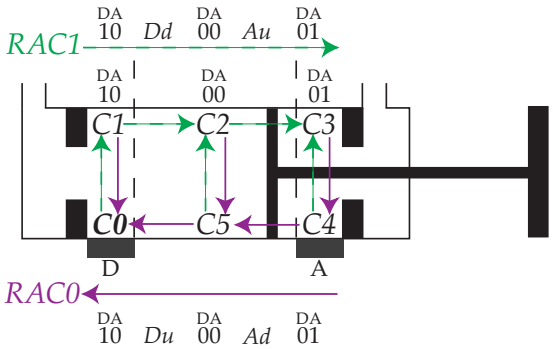


Figure 5.5: Illustrative example: Single acting device.

a control command ACu the device starts to move towards the activated position (event $RAC1$ occurs). As the device leaves its initial position the sensors read 00 until the device reaches the activated position where the sensors read 01. During this movement the device can be in three different states (namely $C1$, $C2$ and $C3$) reflected by the different readings of the sensors. From

each states the device can change the movement direction, for this reason are presented states $C0$, $C4$ $C5$, in these states the sensors value in the device have the same respective readings as in $C3$, $C2$ and $C1$. In figure 5.6 is shown the *Physical Constraint Automaton* (PCA) that models these constraints. The initial state $C0$ corresponds to device in deactivate position and RAC0 movement. During an activation sequence (from state $C0$ to state $C3$) sensors DA evolve in with a well define sequence depending from its structure, sensors evolve from reading 10 to reading 01. From state $C1$ to state $C2$ sensor D evolves from a value 1 to value 0 in according with an event Dd, and from state $C2$ to state $C3$ sensor A evolves from value 0 to value 1 in according with Au.

The PCA automaton captures all the physical connection between the components, it is impor-

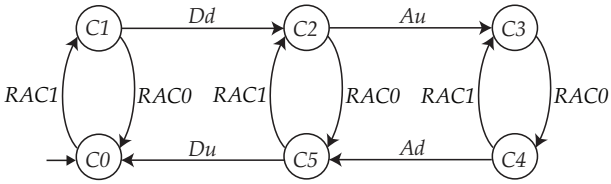


Figure 5.6: Physical Constraint Automaton (PCA) $G_{L,PCA}$

tant to remark events $RAC0$ and $RAC1$ are not control command, but they are the movement of the device, this model is complete independent from control logic. All events in this model are unobservable events, these events interact with sensors and actuator model that generate observable events, as it is shown figure 5.3. Control logic can observe control commands ACu, ACd, and events of sensors value ($RA0$, $RD1$ etc.), but control logic can not see movement of the device, i.e. events $RAC0$ and $RAC1$ are unobservable for control logic. Events Au, Du, etc. are not observable because control logic can only acquire sensor value. The PCA automaton of figure 5.6, is like a component that receive as input the events of the device movement from actuator model, and generate the correct events to change sensors states. In section 5.2.4 this approach to encapsulate unobservable events will help to model the components fault model.

Composing the automata of sensors and actuator in figure 5.4 and PCA automaton in figure 5.6 we obtain the automaton in figure 5.7 (12 states 62 transitions).

Remark The automata models of figure 5.7 do not includes any control logic, but the nominal activation and deactivation sequence of the device that could be implemented by a controller is shown in blue (path from 1 to 8). Of course the automaton can perform other sequence depending on the actual control logic employed, or as will be possible see in next sections (see), in the case due to the occurrence of faults.

Remark The operation of the device is characterized by three sources of information, readings from the two sensors and commands to the actuator. This information indicates the state of the device. The state of the device can be changed by a new control command (events ACu, ACd) that forces a change in the states of the sensors (events Au, Ad, Du, Dd) through the constraint automaton PCA. The self loops of the automata in figure 5.7 and can be thought of as "outputs" emitted from a given state in much the same manner as a Moore automaton.

The state of the device can be change after a new control command, for example in figure 5.8, transition 1 is a transition generates after a new control command ACu from control logic, this transition let's evolve the state from $[RD0\ RA1\ ACd]$ to $[RD0\ RA1\ ACu]$. Also if the device do not change its sensors configuration, the device is in a new state because is changed the control command. This changes will force the actuator to move (event $RAC1$ is generated). This event

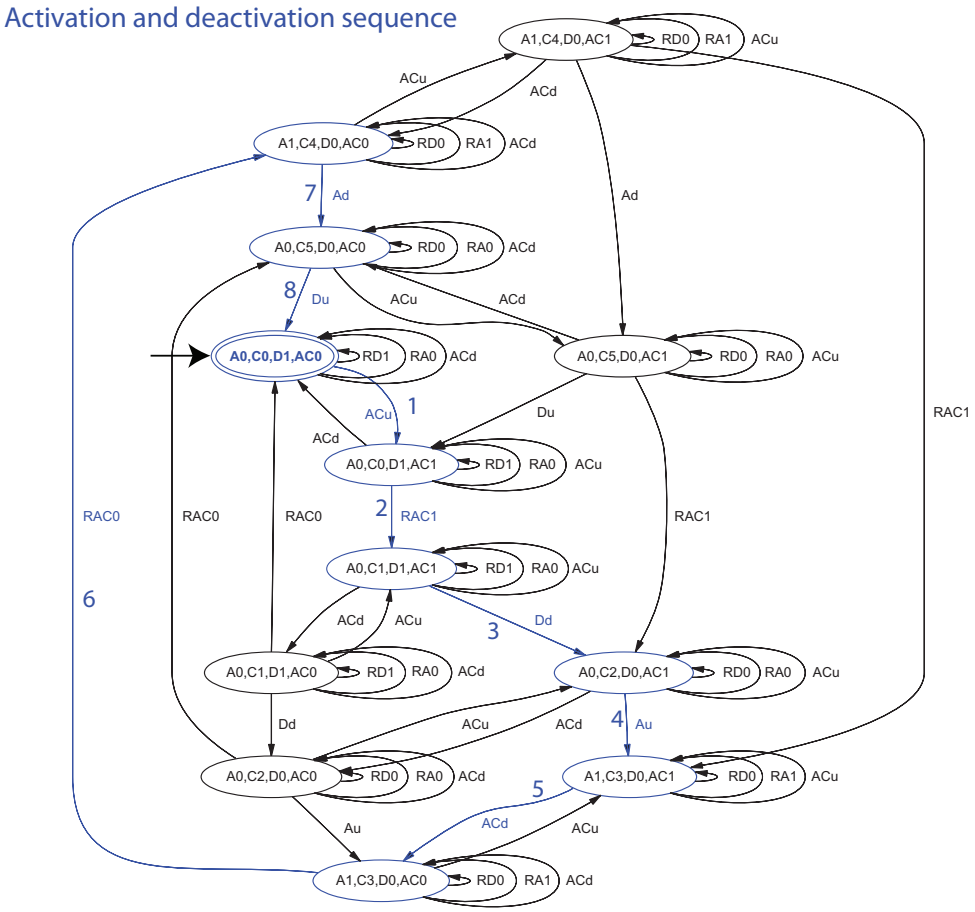


Figure 5.7: Composition of nominal sensors, actuator and PCA. $G_{L,CompNom}$

will force a new change of the state, because the automata connection constrain will generate a Dd event, transition 3, this event generate a sensor change state, so the state of the device evolves form $[RD1\ RA1\ ACu]$ to $[RD0\ RA1\ ACu]$.

5.2.4 Modeling fault at low level

If the process of designing the automaton in figure 5.7 can be considered relatively straightforward, the task becomes significantly more difficult when trying to embed in the model the effect of faults. When happens a fault on a components it is an hard task to know the system behavior after the fault, and a more complicate case is with the occurrence of multiple faults. Following the approach presented here, modeling a fault in a physical component can be accomplished by simply modifying the physical models and the constraint models according to the local effect of the fault.
Consider for example the case in which the activation sensor A, of the device of figure 5.2 can be stuck at low level; we model this fault with with unobservable event f_{a0}. We can model the sensor with fault f_{a0} with automata of figure 5.9(a), if sensor A is stuck low, sensor value will not be able high value, to model this effect from state $A0$, $A1$ when an event f_{a0} happens, the automaton evolves to state $AF0$ where it can no longer change its state. The automata in figure 5.10 is the result of parallel composition of automata of figure 5.9, this automata has in each

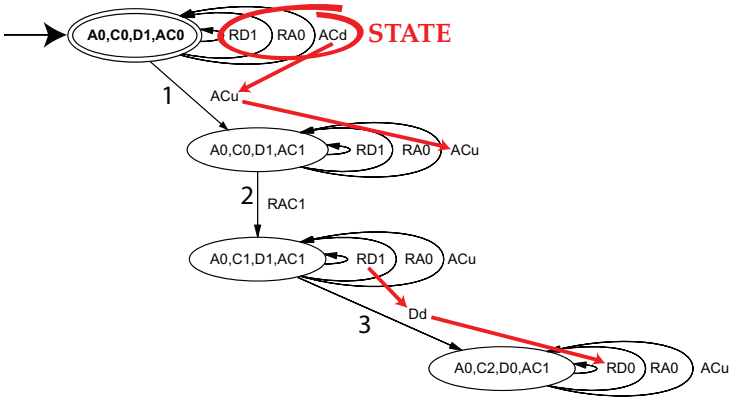
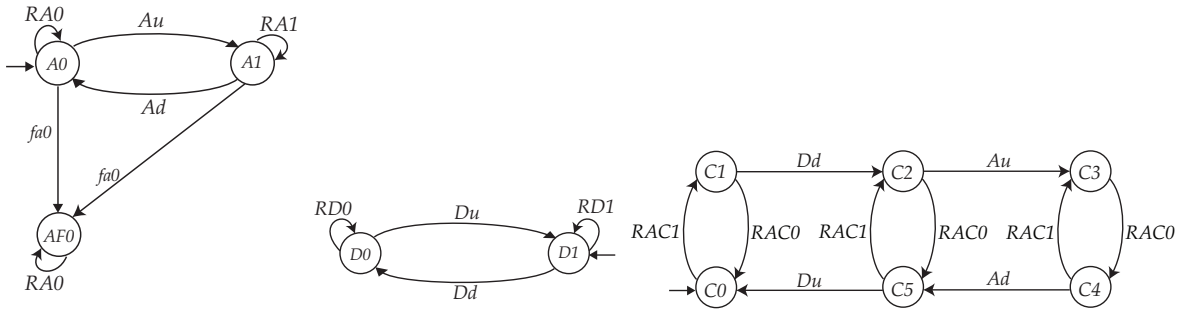


Figure 5.8: States model of the system.



(a) Fault model of sensor A, $G_{L1,Afa0}$.

(b) No fault model of sensor D, $G_{L,Dnf}$.

(c) (PCA) $G_{L,PCA}$.

Figure 5.9: Composition of PCA automata, sensor D and sensor A fault model.

state only two self loop because in this composition we not consider actuator, but only sensors on the parallel composition. The automaton of figure 5.10 is the model of the device without model of actuator, the sequence of state in blue is the sequence of activation and deactivation sequence, in green is depicted the sequence of activation and deactivation when fault f_{a0} happens. It's possible to see that from each state fault f_{a0} can occurred and the system evolves from nominal cycle of activation and deactivation sequence, to faulty cycle of activation and deactivation. In the model there is a livelock, but this livelock is not a physical behavior of the system. When the fault f_{a0} occurs the device can move in the two directions, but sensor A has always low value and sensor D work in normal condition. The livelock is generated because in our model we not considered that when a fault occurs, we do not have only a consequence on the model of sensor, but the fault change the physical interaction between the component. When a sensor is stuck low from a physical point of view is like the device do not have the sensor. In this case if sensor A is stuck low, we can remodeling the device with only sensor D, and the device becomes the device of figure 5.11. The model of this device is composed by an automata of 4 states, because now we can have only two sensors configuration, 10 configuration and 00 configuration, this model can be obtained, as it shown in figure 5.11(b), from model of PCA automaton. If sensor A is stuck low the model has not to generate events Au and Ad, for this reason the model on sensor A when occurs fault f_{a0} evolves and remains in state $AF0$.

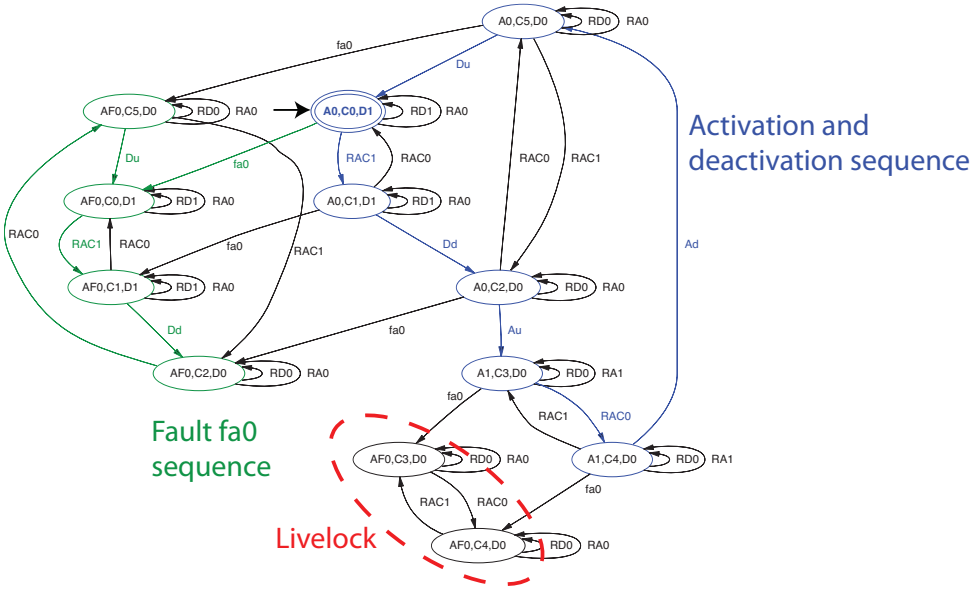
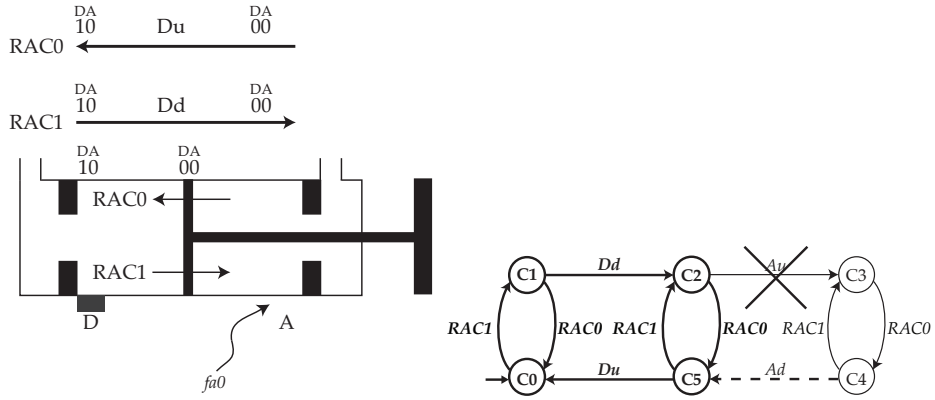


Figure 5.10: Nominal and faulty model with livelock.

The PCA automaton do not reach states $C3$ and $C4$ but evolves only in states $C0$, $C1$, $C2$ and $C5$. If the fault occurs when the system is in states $C3$ or $C4$ the model has a livelock. This livelock is generated, as it is possible to see in figure 5.10, because now event Ad is not generate from sensor model, we have to considerate in the physical constraint automaton the new interaction between component. This new model is depicted in figure 5.12(c). This event ($OutA$) must be considered also in the model of the sensor, for this reason sensor A it is the automata of figure 5.12(a), the parallel composition of automata of figure 5.12 is shown in figure 5.13. The model presented in figure 5.13 is the model of the device in no fault condition and fault condition, the powerful of this approach is that the complete model of the system is generate from parallel composition of simples automata. The effect of the fault can be considered as a local effect on the component when fault happens, in this example the local effect of the fault is the self loop $RA0$ on the sensor model and the event $OutA$ on the sensor model and on connection constrain model it's the effect of the fault on the interaction of the component, the complete behavior of the system is generate by parallel composition of the model, and don't be model, this is the powerful of this approach.

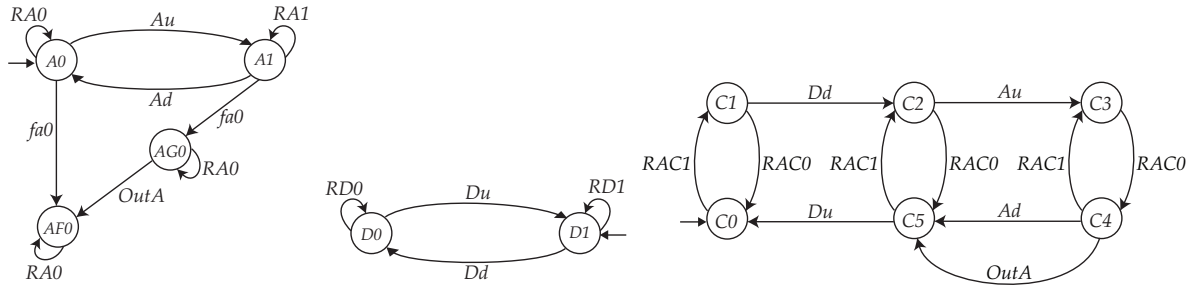
Remark The fault has two consequences, a *local* consequence on the sensor modeled by the automaton figure 5.12(a) getting stuck in state $AF0$ and a *global* consequence on the whole device modeled by PCA in figure 5.12(c)

A sensor can fault in stuck high, in this case the device change its physical configuration as depicted in figure 5.14. This case is similar with the case when sensor is stuck low. The physical constraints are the same because there in no a different relation between the sensors, the different condition fault is on sensor (now is stuck high) and also in this case the model has not to generate events Au and Ad, for this reason the model on sensor A when occurs fault f_{a1} evolves and remains in state $AF1$ from state $A0$ or evolves in state $AF1$ with event $OutA$. The PCA automaton do not reach states $C3$ and $C4$ but evolves only in states $C0$, $C1$, $C2$ and $C5$. With a stuck high fault on sensor A the PCA automaton has the same evolution of sensor stuck low, this because in PCA is captured the information of a sensor fault, and in sensor model is



(a) Mechanical configuration fault f_{a0}. (b) New configuration of (PCA) with fault f_{a0}.

Figure 5.11: Models of fault f_{a0}.



(a) Fault model of sensor A, $G_{L,Afa0}$. (b) No fault model of sensor D, $G_{L,Dnf}$. (c) Physical Constraint Automaton with fault f_{a0}, G_{L,PCA_A}.

Figure 5.12: Composition of automata connection with sensor A fault model.

captured the information if sensor is stuck low or stuck high. In figure 5.15(a) is shown the complete model of sensor A it is possible to note the symmetry structure of the model.
In figure 5.15(b) is depicted the model of sensor D. The model has the same structure of model of sensor A, in this model there is event $OutD$ which the dual means of event $OutA$. The model of how changes physical constraints is reported in figure C.6, and figure C.7 in appendix C. When a fault on sensor D occurred, also the PCA automaton has been model with a new event ($OutD$), the new PCA automaton is depicted in figure 5.16. When sensor D is stuck low (or stuck high), the model of this device is composed by an automata of 4 states, because now we can have only two sensors configuration, 00 configuration and 01 configuration, this model can be obtained, as it shown in figure C.6(b), from model of PCA automaton. If sensor D is stuck low the model has not to generate events Au and Ad, for this reason the model on sensor D when occurs fault f_{d0} evolves and remains in state $DF0$. The PCA automaton do not reach states $C0$ and $C1$ but evolves only in states $C2$, $C3$, $C4$ and $C5$
. In the model presented it is considerate only one fault, but on a device can occur multiple fault, sensor A and sensor D can be fault at the same time. The model building methodology is complete modular and the same model of the components can be applied. The physical constraint automaton has to change because when two fault are possible we introduce a new

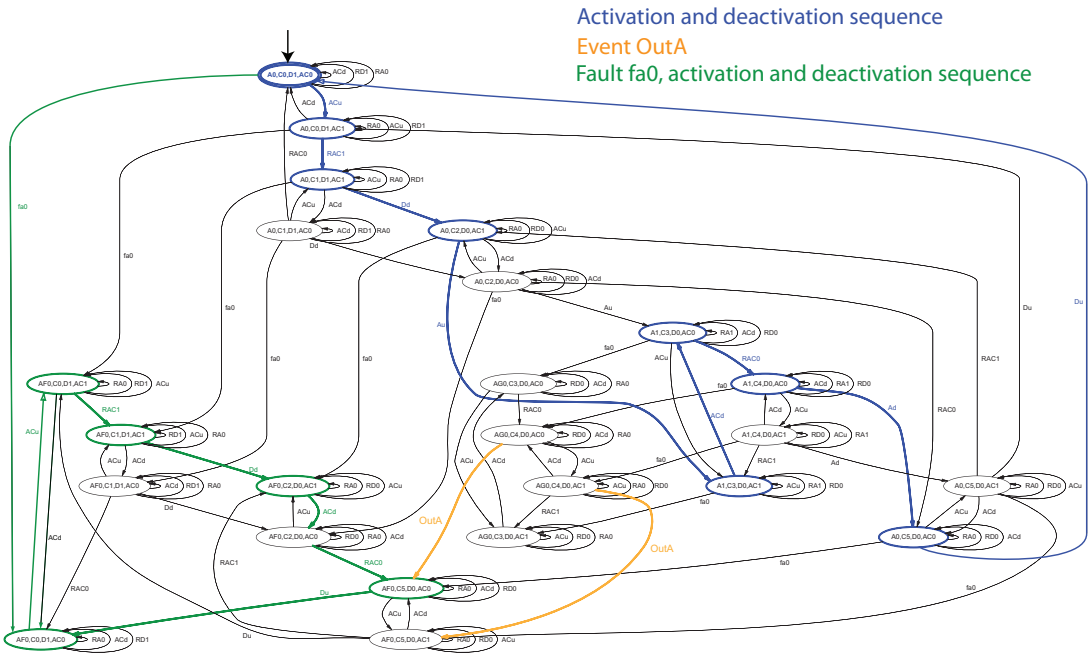
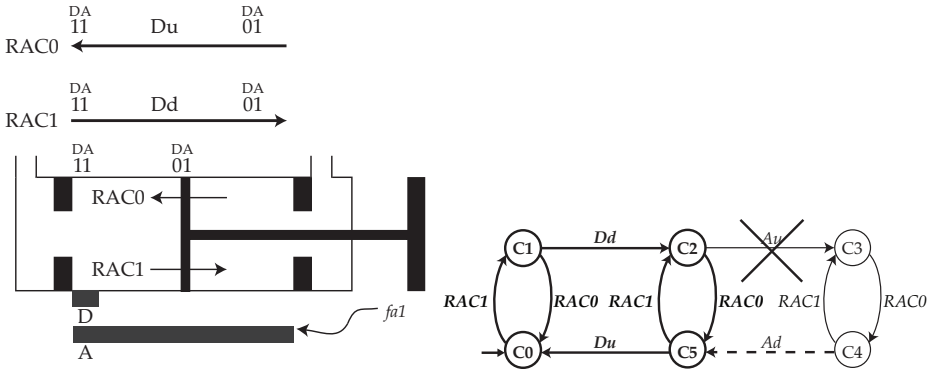


Figure 5.13: Nominal and faulty model for the single acting device, $G_{L,Compfa0}$

physical constraint on the device. In figure 5.17 and figure 5.18 are shown how device change after two faults. When on the device two faults occur the sensor configuration is always stuck at the same value, (11 or 00 or 01 or 10) depending how the sensor are stuck. In this condition the state of the device is depending only from the actuation command and it can evolves only between states $C2$ and $C5$. The new Physical Constraint Automaton that capture this condition is depicted in figure 5.19.
To model a fault on actuator let considerate the actuator is stuck low. When the actuator is stuck low means that the device can moves only in one direction, the control logic can send different command but he device do not correct answer. In figure 5.20(a) is depicted the model of actuator when a fault f_0 occurs, this fault means that the device is stuck to move only in the direction with event $RAC0$. When a fault happens the model of actuator evolves in state $ActF0$ and from this state the actuator can see the different control command but not evolves in new state. In figure 5.20(b) the complete model of actuator is depicted. In B are reported how physical constraints on the device evolves after a fault occurs on actuator. It is easy to understand how a fault on actuator is for the device a constant command from control logic.

5.2.5 Control and monitoring of low level devices

The role of supervisory control logic consists of properly managing the whole system by forcing the desired sequence of actions (at the high level) and by controlling the basic devices to accomplish the desired actions (at the low level). This architecture where the high-level supervisor sends events to the low level in order to force basic actions is depicted in figure 5.21 for the case of a single acting device like the one shown in figure 5.2. More specifically, the event Ra is used to request an activation of the device, while the event Rde is employed to request a deactivation of the device. Note that we use two request events even if the device is single acting, in this way the policy is independent from the device implementation which is

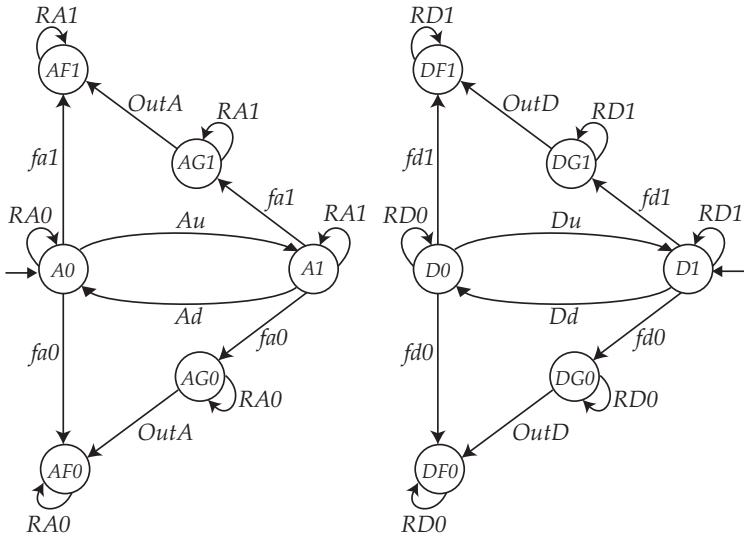


(a) Mechanical configuration fault f_{a1}. (b) New configuration of (PCA) with fault f_{a0}.

Figure 5.14: Models of fault f_{a1}.

hidden in the low-level control logic. When the device has accomplished the high-level request it notifies the high level using event Aa (activation accomplished) and event Ade (deactivation accomplished). The desired behavior for the controlled device is depicted by the automaton $E_{L,ConNom}$ shown in Fig. 5.22; this automaton is the low-level specification. The device is initially inactive, when the high level asks for an activation it sends the event Ra which causes the command ACu to be sent to the actuator. After a given amount of time event $RD0$ signals that the device is not inactive anymore. After an additional amount of time event $RA1$ indicates that the device is activated. This fact causes event Aa that acknowledges the accomplishment of the request to the high level. At this point the low-level control waits for a deactivation request (event Rde). When this request arrives it starts the deactivation cycle which is dual to the activation cycle. When the device is deactivated it generates the event Ade. The observability and controllability of the low level events are as defined previously, while the high level request and answer events $\{Ra, Aa, Rde, Ade\}$ are observable and controllable.
Summarize the controllable and uncontrollable events set are: $\Sigma_{A,c} = \{RA0, RA1\}$, $\Sigma_{D,c} = \{RD0, RD1\}$, $\Sigma_{Act,c} = \{ACu, ACd\}$ $\Sigma_{Con,c} = \{Ra, Aa, Rde, Ade, RA0, RA1, RD0, RD1, ACu, ACd\}$ and the following uncontrollable event sets $\Sigma_{A,uc} = \{Au, Ad, fa0, \}$, $\Sigma_{D,uc} = \{Du, Dd\}$ and $\Sigma_{Act,uc} = \{RAC0, RAC1\}$
Composing the low-level specification in figure 5.22 with the nominal sensors, actuator and PCA models of figure 5.4 and figure 5.6, we obtain the controlled device model $G_{L,DevNom}$ shown in figure 5.23(a). It is possible to note that figure 5.23(a) the automaton doesn't have self loop in the states, instead model without control in each states has self loop. From a physical point of view the model without control is the model of the device, in each states the device has a sensors configuration, and the last command sends to actuator, the control logic can acquire the information on the sensors, and information on the last command send to actuator in each time. Let's considered this part of code of control logic:

```
CASE DeviceState OF
Deactive:
IF (Ractive) THAN
Actuator:=TRUE;
DeviceState:=InActivation;
END_IF;
```



(a) Fault Model of sensor A, $G_{L,Afa}$.

(b) Fault Model of sensor D, $G_{L,Dfd}$.

Figure 5.15: Models of sensor A and sensor D.

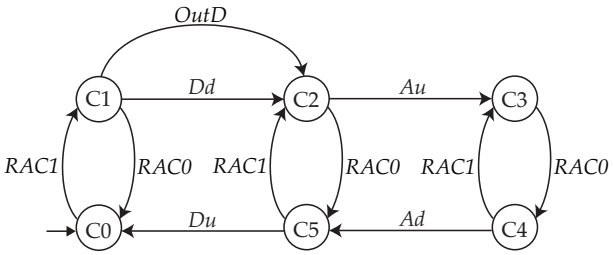
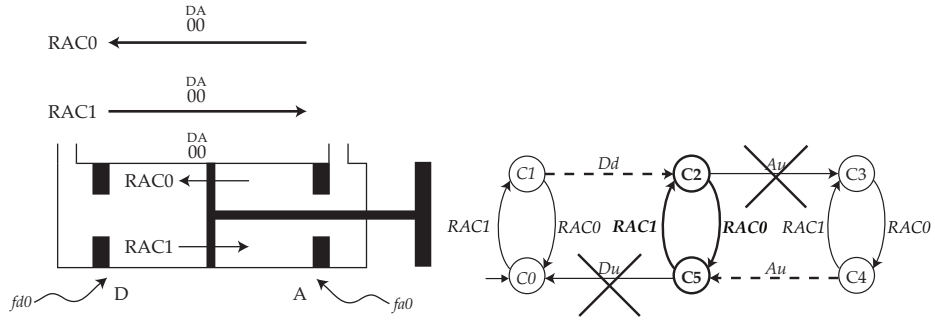


Figure 5.16: Physical Constraint Automaton with fault f_{d0} or fault f_{d1}, G_{L,PCA_D}.

```
.
.
IF (SensorA) THAN
Aactive:=TRUE;
DeviceState:=Activated;
END_IF;
.
.
END_
```

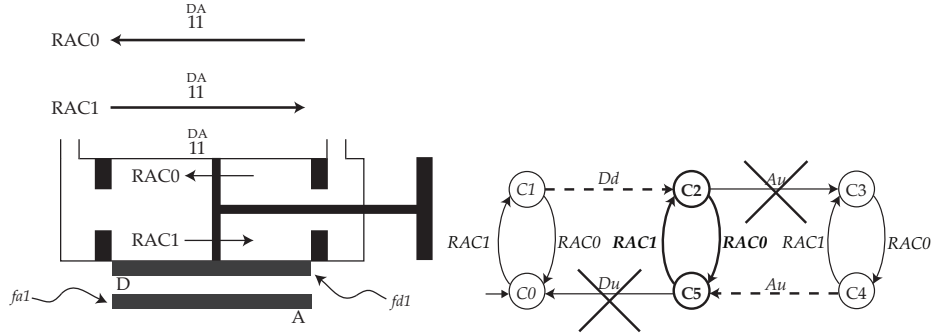
The construct `CASE` implements the automata, the variable DeviceState is the state of control automata, when it executes parallel composition between control and model of device is like the physical operation that control acquire for example a sensor value, the control can read in any time a sensor value, this feature is implemented in the model by self loop with sensor configuration in each state. For example when control executes instruction `IF (SensorA) Then` the control loops on acquire sensor A value, and when it's value is high the control executes the code in `THAN` instruction, from a point of view of model this is equivalent to model control



(a) Mechanical configuration fault f_{d0} and f_{a0}.

(b) New configuration of (PCA) with faults f_{d0} and f_{a0}.

Figure 5.17: Models of fault f_{d0} and f_{a0}.



(a) Mechanical configuration fault f_{d1} and f_{a1}.

(b) New configuration of (PCA) with faults f_{d1} and f_{a1}.

Figure 5.18: Models of faults f_{d1} and f_{a1}.

is in a state that generate event $RA1$ and device model evolves in a state with a self loop event $RA1$, when control and device model are both in a state with $RA1$ events, parallel composition generates event $RA1$ in complete model, and this event in complete model corresponding to sensor A value is high and control executes the THAN.

The process of determining whether the control desired by $E_{L,ConNom}$ is actually achievable first requires defining the automaton $G^{lifted}_{L,CompNom}$ as $G_{L,CompNom}$ with self-loops added for events in the set $\Sigma_{E_{L,ConNom}} \backslash \Sigma_{G_{L,CompNom}} = \{Ra, Aa, Rde, Ade\}$. The language generated by $G_{L,DevNom}$, $\mathcal{L}(G_{L,DevNom}) \subseteq \mathcal{L}(G^{lifted}_{L,CompNom})$, can be shown to be controllable and observable with respect to $\mathcal{L}(G^{lifted}_{L,CompNom})$. Therefore, there exists a supervisor S such that $S/G^{lifted}_{L,CompNom} = G_{L,DevNom}$. $E_{L,ConNom}$ is an automaton realization of supervisor S.

If we consider the models for the single acting device with fault $fa0$ (see fig. 5.12) composed with the nominal controller in fig. 5.22, we obtain the controlled device model $G_{L,Devfa0}$ in fig. 5.23(b) from the parallel composition of models $G_{L,Afa0}, G_{L,Dnf}, G_{L,Actnf}, G_{L,PCA\_A}$ and $E_{L,ConNom}$. Note that the automaton $G_{L,Devfa0}$ in figure 5.23(b) has a deadlock due to the occurrence of the fault $fa0$. Since sensor A cannot rise to true, this deadlock arises due to the fact that the controller will never know when the device has been activated. To avoid this deadlock we consider a new logical model that considers also timing constraints. When the device

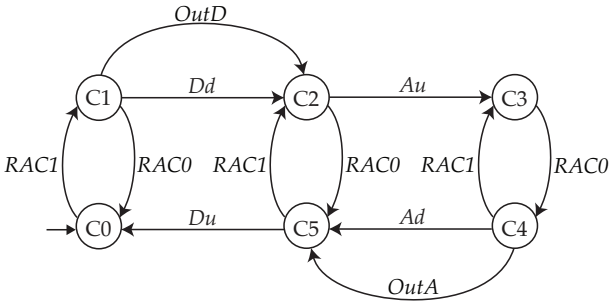
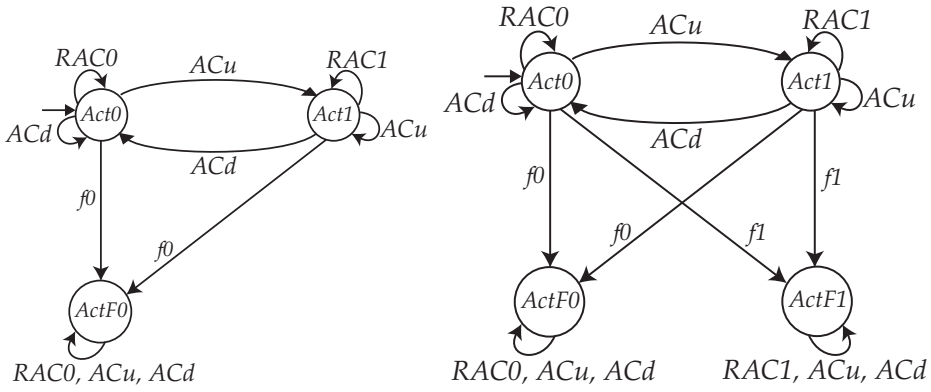


Figure 5.19: Physical Constraint Automaton with fault on sensor A and sensor D.



(a) Fault Model of actuator $G_{L,Actf0}$.

(b) Fault Model of actuator $G_{L,Act}$.

Figure 5.20: Models of actuator faults.

is moving from one steady state position to the other, supposing that we have some estimate of the transient timing, we can detect the occurrence of a fault and take some remedial actions when the operation is not finished within the expected time. The aim of this strategy is twofold: (i) embed basic fault diagnosis in the low-level control, leaving to the high level only the diagnostic task of detecting complex faults that involve multiple devices; and (ii) avoid deadlock due to faults. Following this idea, we want to introduce into the model of the device the following temporal rule: having in mind the desired behavior of the controlled device depicted in figure 5.22, we suppose that the amount of time needed to activate the device (to evolve from state $S0$ to state $S5$ in figure 5.22) and the amount of time needed to deactivate the device (to evolve from state $S5$ to state $S0$) is bounded by some known amount when the device is operating correctly. The diagnostic logic will check the consistency of this rule during the evolution of the device to determine whether or not a fault has occurred. Such a diagnostics algorithm will not require the introduction of timed automata as the necessary timing information can be captured by a timeout event TO that signals the violation of the deadline. With this in mind we consider the timer model in figure 5.24. Event SC is used to start the timer, while event RTO is used to reset the timer. Note that the timeout event TO can only occur after the occurrence of the fault $fa0$ because this is supposed to be the only case in which the temporal rule is violated. Note that SC, RTO and TO are observable and controllable events. In order to avoid deadlock in the controlled faulty device, we substitute the nominal super-

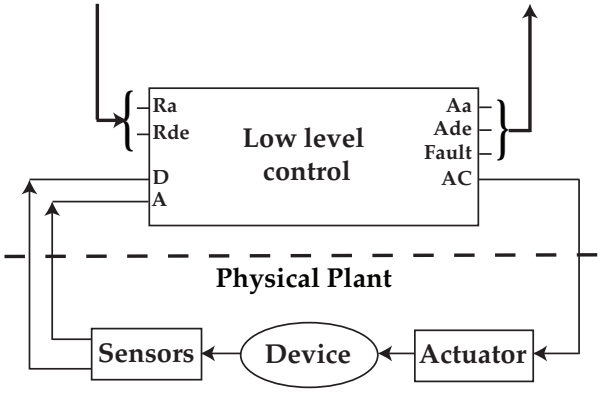


Figure 5.21: Single actuator device control.

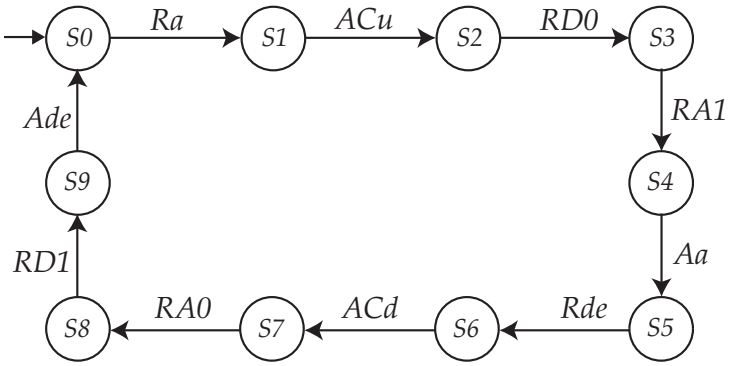
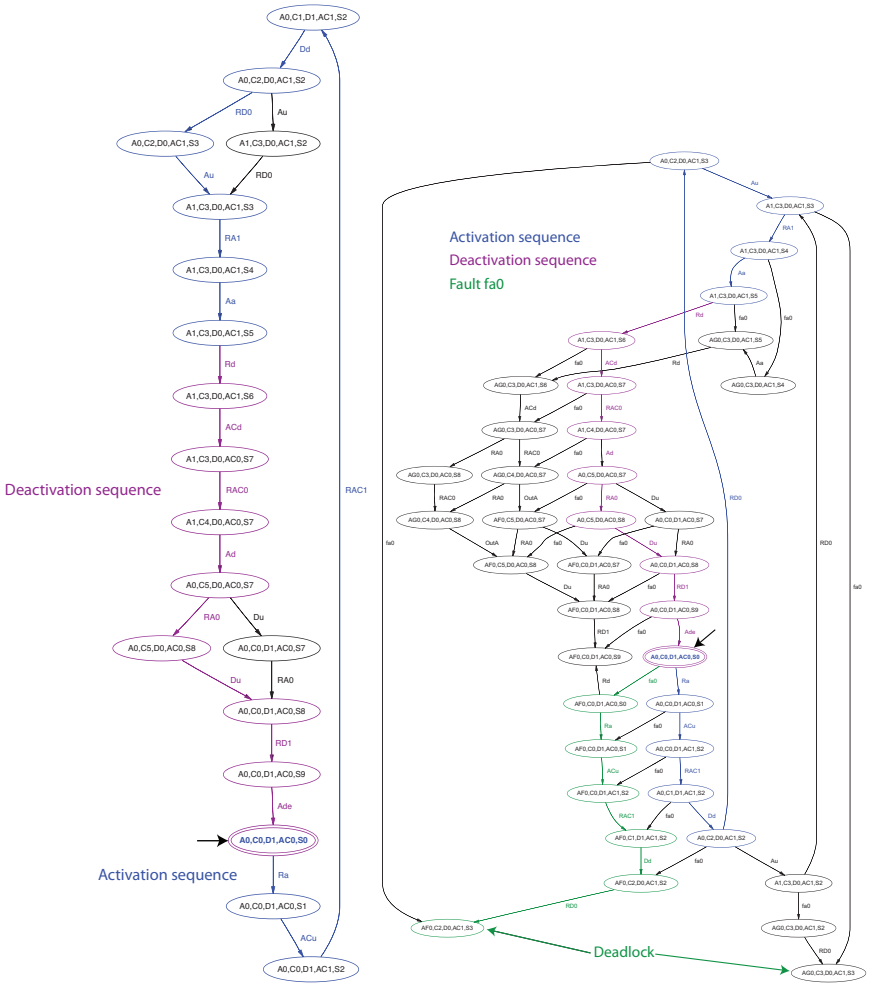


Figure 5.22: Specification automaton $E_{L,ConNom}$ for low level control of a single acting device.

visor $E_{L,ConNom}$ shown in figure 5.22 with the supervisor $E_{L,ConDiag}$ depicted in figure 5.25. $E_{L,ConDiag}$ is generated by designer understanding of the system and not by formal synthesis, but the properties of observability and controllability were verified formally in the same manner as before for $E_{L,ConNom}$. The new supervisor embeds not only control like $E_{L,ConNom}$ did, but also fault detection. We further consider notions of *Dynamic fault detection* and *Static fault detection*. If the device is in the activated position waiting for a request of deactivation (state $S7$) and the sensor A spontaneously changes its output to low, the fault $fa0$ has occurred and it is detected when event $RA0$ occurs taking the supervisor to state $S15$. When a fault is detected without a movement of the system we classify it as static fault detection, shown by point-dashed lines in figure 5.25. If the fault $fa0$ occurs when the sensor value of A is low, the fault can only be detected when a movement of the device occurs which should force the value A to high. For this reason when the supervisor receives a request of activation (event Ra), this causes not only the event ACu but also the event SC that starts the timer. In fact in this case the device never reaches the activated position because the event $RA1$ never occurs and so the control does not reset the timer (event RTO). The fault is then detected in state $S14$ by the event TO that signals the violation of the time deadline. We classify this as dynamic fault detection indicated by dashed lines in figure 5.25.

Composing the new low-level supervisor in figure 5.25 and the timer model in figure 5.24 with the sensors, actuator and PCA model, we obtain the controlled device model $G_{L,Totfa0}$



(a) Controlled nominal single acting device model, $G_{L,DevNom}$.

(b) Controlled faulty single acting device model, $G_{L,Devfa0}$.

Figure 5.23: Controlled single acting device models.

from parallel composition of models: $G_{L,Afa0}, G_{L,Dnf}, G_{L,Actnf}, G_{L,PCA\_A}, E_{L,ConDiag}$ and $G_{L,Tfa0}$; the resulting automaton has 57 states and 94 transitions; it is not shown here. Note that now, even after a fault, the controlled device does not deadlock. At this point it is possible to use the automaton $G_{L,Totfa0}$ to build a diagnoser and check the diagnosability properties of the controlled device with the proposed supervisor. The diagnoser is shown in figure 5.26; as it does not contain any indeterminate cycles, the fault fao is diagnosable. Examining the diagnoser in figure 5.26 and the supervisor in figure 5.25, we can see that we can use the supervisor to detect faults based on its entry into state S14 or state S15. The meaning of state S15 is clear: event TO has occurred before the activation request was carried out. State S14 is entered when sensor A changes its reading from high to low without a deactivation request. We can confirm using the diagnoser in figure 5.26 that whenever the supervisor enters S14 or S15, the diagnoser is indeed in a fault-certain state. Moreover, all cycles of certain states in the diagnoser visit either S14 or S15. Therefore, the low-level supervisor we have constructed can be employed for both control and diagnostic purposes and the actual diagnoser does not need to be stored in memory. The

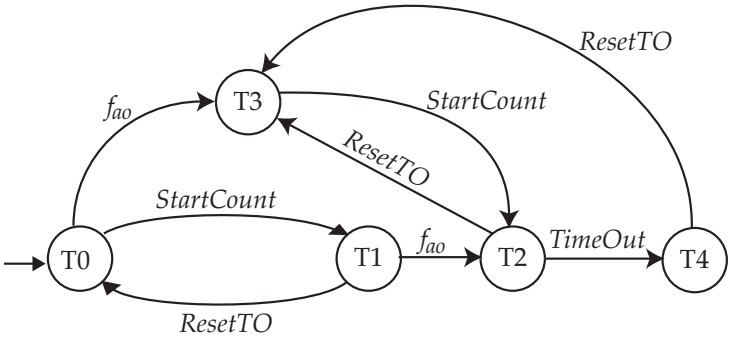


Figure 5.24: Timer model for fault $fa0$, $G_{L,Tfa0}$

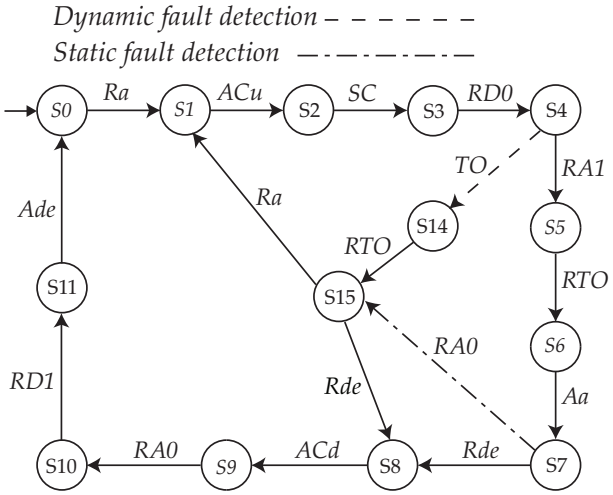


Figure 5.25: Supervisor $G_{L,ConDiag}$ for the single acting device considering fault $fa0$.

complete state concatenation of diagnoser is reported in C.2. On the base of teh **Algorithm:** Definition set of diagnosis controller state SD, in the example SD={S14, S15}

Procedure to proof embed control diagnosis detect fault.

Step 1: Build complete faulty model: G

Step 2: Test diagnosability, if $Diag(G)$ is diagnosable go to step 3.1

Step 3: Examination $Diag(G)$

Step 3.1: Controller component of G is SD $\rightarrow$ $Diag(G)$ is in F-certain state.

Step 3.2: $Diag(G)$ is in F-uncertain state or normal state $\rightarrow$ controller component G is not in SD.
$x_d \in Diag(G)$ is so called on DG state (diagnoser controller state. If $\forall x \in x_d$, the controller component of x is in SD.)

Step 3.3: $\forall$ FEC in $Diag(G)\nexists$ arbitrary long suffix that does not visit a DG-state.

5.3 Conclusions on DES approach for formal verification

In this chapter it was presented a general approach to discrete-event modeling of physical behavior and control logic in industrial automaton. The key features of the proposed approach are its modularity, exploiting parallel composition to obtain the complete system model from that of individual components, and the reusability of the generic component models. The reusability of component models is made possible by the construction of a so-called "physical constraint automaton" that captures the physical coupling of generic components in a given automated system. We first build fault-free models then show how to extend them to include faulty behavior, preserving modularity. We employ a hierarchical decomposition that separates the control logic into low-level control actions and high-level control actions, coupled through an interface. With this methodology it is possible to formal verify the algorithm of fault detection proposed in chapter 4 on the architecture of *Generalized Device*
In this work was proposed an example based on a single acting cylinder, it is possible to extend the procedure to a double acting cylinder changing only the physical constraint automaton as it is possible to see in figure 5.27. The new event $RACS$ it is the event when the device is stopped, but all other events are the same an interact with sensor A and sensor D in the same manner. With the same criteria it is possible to do an extension to modeling of electric motor like the rotary table of FESTO (see B). In this case an electric motor used to do a positioning operation has one sensors, for example sensor D. In this case the motor can move in clockwise or unclockwise and this movement are events $RAC0$ and $RAC1$. When an event $RACS$ occurs the motor is stopped. The high level of the proposed architecture is the sequence of actions proposed in chapter 3 and chapter 4 and the interface, speaking roughly is the link between the high level action and the activation and deactivation request of low level. An important point, it will development in future work it analyze the diagnosability of the entire systems (the entire machine) as compositon of the diagnosability information of the components. In this work was development fault detection on sensors and actuator fault, but it is not take in consideration mechanical fault on the systems. The idea is a mechanical system has the same "effect ", from a point of view of the control system, of a fault on sensor or actuator or a combination of twice. it should be interesting find a partition with different kind of fault, mechanical faults, sensor faults etc. with the same partition of symptoms.

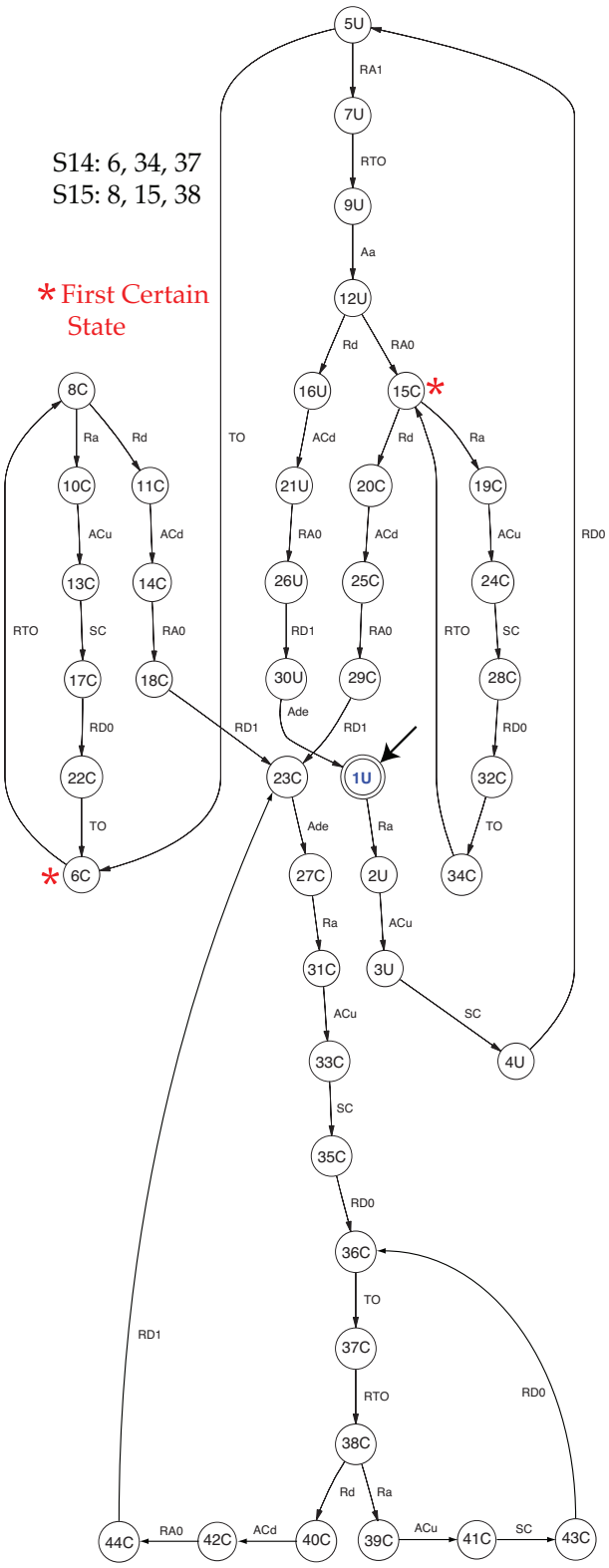


Figure 5.26: Diagnoser of the closed loop model $G_{L,Totfa0}$.

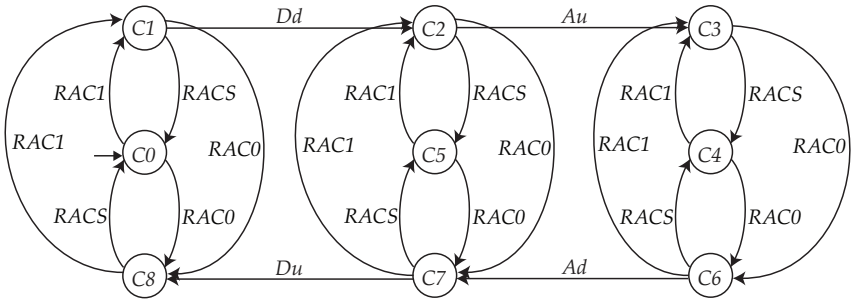


Figure 5.27: Physical Constraint Automaton for a double acting cylinder.

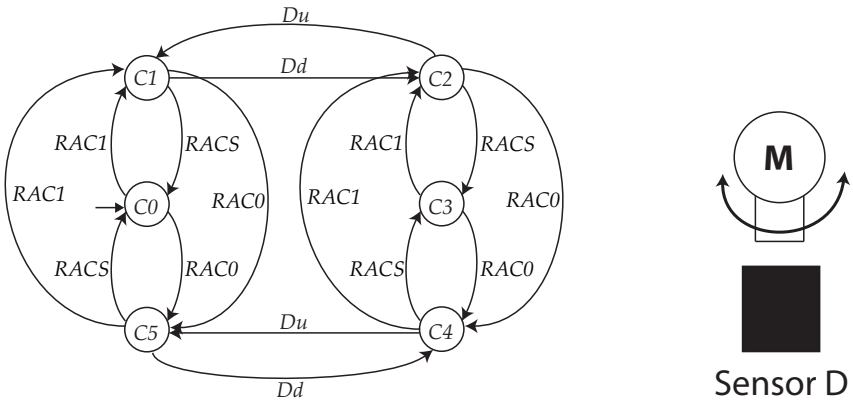


Figure 5.28: Physical Constraint Automaton for a electric motor.

5.4 Active fault tolerant control online diagnostics

In this section is reported some research result on the problem of Fault Tolerant Control in the framework of Discrete Event Systems modeled as automata. A fault tolerant controller is a controller able to satisfy control specifications both in nominal operation and after the occurrence of a fault. This task is solved by means of a parameterized controller which is suitably updated on the basis of the information provided by on-line diagnostics: the supervisor actively reacts to the detection of a malfunctioning component in order to eventually meet degraded control specifications. Starting from an appropriate model of the system, we recall the notion of safe diagnosability as a necessary step in order to achieve fault tolerant control. We then introduce two new notions: (i) "safe controllability", which represents the capability, after the occurrence of a fault, of steering the system away from forbidden zones and (ii) "active fault tolerant system", which is the property of safely continuing operation after faults. Finally, we show how the problem can be solved using a general control architecture based on the use of special kind of diagnoser, called "diagnosing-controller", which is used to safely detect faults and to switch between the nominal control policy and a bank of reconfigured control policies.

5.4.1 Fault tolerant control

Complex technological systems are vulnerable to unpredictable events that can cause undesired reactions and as a consequence damage to technical parts of the plant, to personnel, or to the environment. The main objective of the Fault Detection and Isolation (FDI) research area (see, e.g., [73]) is to study methodologies for identifying and exactly characterizing possible incipient faults arising in predetermined parts of the plant. This is usually achieved by designing a dynamical system which, by processing input/output data, is able to detect the presence of an incipient fault and eventually to precisely isolate it. Once a fault has been detected and isolated, the next natural step is to reconfigure the control law in order to tolerate the fault, namely, to guarantee pre-specified (eventually degraded) performance objectives for the faulty system. In this framework, the FDI phase is usually followed by the design of a Fault Tolerant Control (FTC) system, namely, by the design of a reconfiguring unit that, on the basis of the information provided by the FDI filter, adjusts the controller in order to achieve the prescribed performance for the faulty system (see [6]).

The FTC problem can be tackled using either a passive approach or an active one. The *passive approach* deals with the problem of finding a general controller able to satisfy control specifications both in nominal operation and after the occurrence of a fault. Passive fault tolerance uses robust control techniques to ensure that the closed loop system remains insensitive to certain failures so that the impaired system continues to operate with the same controller and system structure. The effectiveness of the scheme depends upon the robustness of the nominal fault-free closed loop system. Hence, a unique controller, designed off-line, can be used and on-line fault information is not required. In contrast, *active fault tolerance* aims at achieving the control objectives by adapting the control law to the faulty system behavior. In general, the latter phase is carried out by means of a parameterized controller which is suitably updated by a *supervisory unit*, on the basis of the information provided by the FDI filter. This approach relies upon a "certainty equivalence"idea extensively used in the context of adaptive control, since it is based on the explicit estimation of faults by the FDI filter and subsequent explicit reconfiguration of the controller in presence of faults.

In this paper we consider the FTC problem for systems that are governed by operational rules that can be modeled by Discrete Event Systems (DES), i.e., dynamical systems with dis-

crete state spaces and event-driven transitions. Several methodologies have been developed to solve the FDI problem for systems modeled as DES; see [20], [38], [52], [61], [69], [78], [80], for a sample of this work including references to successful industrial applications. Less effort however has been spent to solve the FTC problem in the DES framework; this problem has recently been studied in [24], [95], [96] and [97]. In [24] the problem of managing a set of real-time periodic tasks into a set of processors upon the occurrence of a fault (considered as observable) on one or more processors is solved using optimal discrete controller synthesis techniques. In [48] the supervisory control technique for Petri nets based on place invariants is adapted to achieve robustness properties for systems in which faults and reconfigurations are modeled as changes in marking. In [97], the authors propose a definition of fault tolerance based on the DES notions of language equivalence and convergence by means of control. Roughly speaking, a DES is said to be fault-tolerant if every post-fault behavior is equivalent to a non-faulty behavior in a bounded number of steps; moreover a supervisor is said to be a "fault-tolerant controller"if it is able to force fault tolerant behavior for the supervised DES. The authors provide a necessary and sufficient condition for the existence of a fault-tolerant supervisor able to enforce a specification for the non-faulty plant and a wider specification for the overall plant. Such an approach can be therefore cast in the framework of passive approaches. We study the active approach to FTC for DES modeled as automata. Specifically, we want to design an architecture in which the supervisor actively reacts to the detection of a malfunctioning component in order to meet eventually degraded control specifications. To this aim we describe a modeling procedure that results in a structured model of the controlled system containing a nominal part and a set of faulty parts. Starting from this suitable model, we recall the notion of safe diagnosability (see [68]) as a necessary step in order to achieve fault tolerant supervision of DES. We then introduce the new notion of *safe controllability*, which represents the capability, after the occurrence of a fault, of steering the system away from forbidden zones. We also define the new notion of *active fault tolerant system* with respect to post-fault specifications as the property of safely continuing operation after faults. We then present a general control architecture to deal with the FTC problem. This architecture is based on the use of special kind of diagnoser, called "diagnosing-controller,"which is used to safely detect faults and to switch between the nominal control policy and a bank of reconfigured control policies. In this sense, the exploited paradigm is that of switching control in which a high-level logic is used to switch between a bank of different controllers (see[39] and [100]).

The main contributions of this work are:

1. the exploitation of a multiple supervisor architecture to actively counteract the effect of faults;
2. the evaluation of the effect of the diagnostics algorithm on the performance of the architecture;
3. the definition of new diagnoser called *diagnosing-controller* which realizes in a unique entity the switching architecture.

5.4.2 Supervisory control of DES with faults

Following the theory of supervisory control of DES (see, e.g., A.1 and [17]), the system is modeled by automaton $G = (X, E, \delta, x_0)$ where X is the state space, E is the set of the events, δ is the partial transition function and x_0 is the initial state of the system. The behavior of the system is described by the prefix-closed language $\mathcal{L}(G)$ generated by G. The event set E is

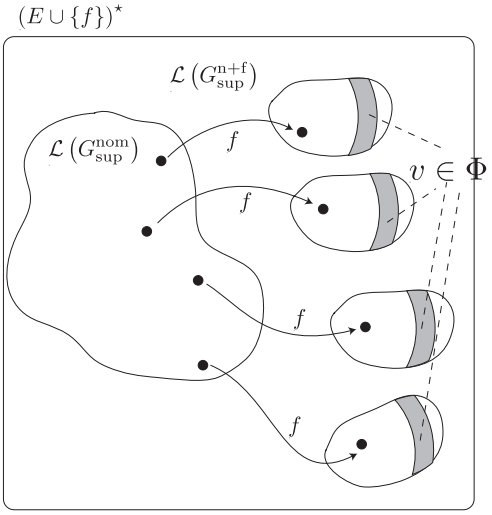


Figure 5.29: Supervised DES with faults.

partitioned as $E = E_o \cup E_{uo}$ where E_o represents the set of observable events (their occurrence can be observed) and E_{uo} represents the set of unobservable events. We associate with E_o the (natural) projection P_{o}, $P_{\mathrm{o}} : E^* \to E_o^*$. Moreover, some of the events are controllable (it is possible to prevent their occurrence) while the rest are uncontrollable. Thus the event set can also be partitioned as $E = E_c \cup E_{uc}$.

We start with the model of the uncontrolled system, denoted by G^{nom} and given in the form of an automaton, and a set of specifications on the controlled behavior. In general G^{nom} is expressed as the interconnection, via parallel composition, of a set of interacting components whose models are denoted by $(G_1^{\mathrm{nom}}, \ldots, G_n^{\mathrm{nom}})$. The behavior of G^{nom}, captured by the language $\mathcal{L}(G^{\mathrm{nom}})$, must be restricted by control in order to satisfy the set of specifications. For this purpose, we design a *supervisor*, whose realization as an automaton is denoted by S^{nom}, and connect it with G^{nom} thereby obtaining the controlled system $G^{\mathrm{nom}}_{\mathrm{sup}} := G^{\mathrm{nom}} \parallel S^{\mathrm{nom}}$ with its associated language $\mathcal{L}(G^{\mathrm{nom}}_{\mathrm{sup}})$ satisfying the set of language specifications $\mathscr{K}^{\mathrm{nom}}$.

Potential faults of the system components are usually considered at this point. In this regard, the G_i^{nom} component models are enhanced to include most likely faults and subsequent faulty behavior (see, e.g., [78]). Therefore, instead of the nominal model G^{nom}, we now have model $G^{\mathrm{n+f}}$ that embeds the (potential) faulty behavior of the respective components. In the following, for the sake of simplicity, we consider a single fault event f; we denote its associated fault type by "F". It follows that $\mathcal{L}(G^{\mathrm{n+f}}) \supset \mathcal{L}(G^{\mathrm{nom}})$ with corresponding event sets $E^{\mathrm{n+f}} = E \cup \{f\}$, where $f \in E_{uo}^{\mathrm{n+f}} \cap E_{uc}^{\mathrm{n+f}}$, i.e., f is unobservable and uncontollable. This means that the actual controlled behavior of the system is described by $G^{\mathrm{n+f}}_{\mathrm{sup}} = G^{\mathrm{n+f}} \parallel S^{\mathrm{nom}}$. This situation is depicted in Fig. 5.29: the structure of $G^{\mathrm{n+f}}_{\mathrm{sup}}$ contains the "nominal part"and a set of "faulty parts". Since we are considering persistent faults, after any occurrence of fault f, the supervised system continues evolving according to well-defined *post-fault models* that are completely disjoint from the nominal supervised model.

By construction of S^{nom}, there are no undesired actions in the nominal part. However, undesired sequences of actions can arise in post-fault models due to the effective control actions of the nominal supervisor on faulty components, as captured in $G^{\mathrm{n+f}}_{\mathrm{sup}}$. Consequently, we must avoid that after fault f occurs, the system executes a forbidden substring from a given finite set

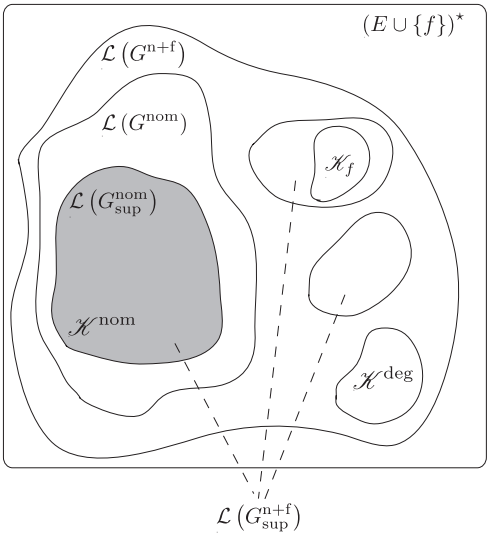


Figure 5.30: Fault Tolerance specifications for a supervised DES.

Φ, where $\Phi \subseteq E^\star$. In essence, the elements of the set Φ capture sequences of events that become illegal after the occurrence of fault f. This situation can be formalized by defining the "illegal language" $\mathscr{K}_f$ as in [68]:

$$\mathscr{K}_f = \left\{ u \in \mathcal{L}(G^{\mathrm{n+f}}_{\mathrm{sup}}) \text{ s.t. } [u = st]\ [s \in \Psi(f)]\ [\exists v \in \Phi \text{ s.t. } v \text{ is a substring of } t] \right\} . \tag{5.6}$$

The language of system $G^{\mathrm{n+f}}$ is divided in two parts: the "nominal part" (corresponding to G^{nom}) and the "faulty part". The faulty part includes the illegal language $\mathscr{K}_f$ which contains all the possible continuations after fault f that have a forbidden string from set Φ as substring. Under the supervision of S^{nom}, the resulting system behavior $\mathcal{L}(G^{\mathrm{n+f}}_{\mathrm{sup}})$ will contain the nominal controlled behavior $\mathcal{L}(G^{\mathrm{nom}}_{\mathrm{sup}})$, which coincides with the nominal specifications $\mathscr{K}^{\mathrm{nom}}$, and in addition post-fault behavior that may include strings that are in the illegal language.

The design objectives of a fault tolerant supervision system can therefore be enumerated as follows:

A) Diagnose the occurrence of event f before the system executes some illegal sequence in the set Φ;

B) Force the system to stop its evolution before the execution of forbidden sequences;

C) Steer the faulty system behavior in order to meet new (eventually degraded) post-fault specifications that are assumed to be expressed in the form of language $\mathscr{K}^{\mathrm{deg}}$.

Figure 5.30 depicts the described scenario from a language specification point of view.

Note that objective **A** is achieved if the property of *safe diagnosability* described in [68] is satisfied by the system. In the following, objective **B** will be studied in terms of a new property called *safe-controllability*, while objective **C** will be linked with the new property of *active fault tolerance*. It is important to emphasize that the post-fault specifications $\mathscr{K}^{\mathrm{deg}}$ are in general disjoint from $\mathcal{L}(G^{\mathrm{n+f}}_{\mathrm{sup}})$; therefore, in order to satisfy them, it is necessary to switch from the nominal supervisor S^{nom} to a new supervisor denoted by S^{deg}, thereby following an active approach to fault tolerance in the sense of [6].

5.4.3 Safe controllability of DES

This section is concerned with the definition and testing of the property of safe controllability for the purpose of the fault tolerance objectives described in the preceding section. First, we recall the definition of diagnosability, introduced in [78], which states that a language L is diagnosable if it is possible to detect within a finite delay occurrences of faults using the record of observed events.

Definition 5.1 [Diagnosable DES] *A prefix-closed language L that is live and does not contain loops of unobservable events is said to be diagnosable with bound n with respect to projection P_{o} and fault event f if the following holds:* $(\exists n \in \mathbb{N})\ (\forall s \in \Psi(f))\ (\forall t \in L/s)\ (\|t\| \geq n \Rightarrow \mathcal{D})$ *where the diagnosability condition $\mathcal{D}$ is:* $\omega \in P_{\mathrm{o}}^{-1}\left[P_{\mathrm{o}}(st)\right] \cap L \Rightarrow f \in \omega$.

Objective **A** of the preceding section requires that after a fault f occurs, the system should not execute a forbidden substring from a given finite set Φ. This objective is captured by the property of safe diagnosability introduced in [68] and now recalled.

Definition 5.2 [Safe Diagnosable DES] *A prefix-closed language L that is live and does not contain loops of unobservable events is said to be safe diagnosable with respect to projection P_{o}, fault event f and forbidden language $\mathscr{K}_f$ if the following conditions hold:*

SC1) Diagnosability condition: L is diagnosable with bound n, with respect to P_{o} and f;

SC2) Safety condition: $(\forall s \in \Psi(f))(\forall t \in L/s)$ *such that* $\|t\| = n$, *let* t_c, $\|t_c\| = n_{t_c}$, *be the shortest prefix of t such that $\mathcal{D}$ holds, then* $\overline{st_c} \cap \mathscr{K}_f = \emptyset$.

In words, this definition says that a language is safe diagnosable if it is diagnosable and if after a fault, the shortest continuation that assures the detection of the fault does not contain any illegal substring from the set Φ.

We make use of the Diagnoser Approach described in [78] to test diagnosability and safe diagnosability. The *diagnoser*, denoted by G^{diag}, is an automaton built from the system model $G_{\mathrm{sup}}^{\mathrm{n+f}}$. This automaton is used to perform diagnosis when it observes on-line the behavior of $G_{\mathrm{sup}}^{\mathrm{n+f}}$. The construction procedure of the diagnoser can be found in [78]. We recall here a theorem from [78] for testing (off-line) the diagnosability of a system using its diagnoser; the reader is referred to [78] for undefined terminology.

Theorem 5.1 (see [78] for details.) *A language L is diagnosable with respect to the projection P_{o} and fault event f if and only if the diagnoser G^{diag} built starting from any generator of L has no F-indeterminate cycles.*

By slightly modifying the diagnoser as explained in [68], we obtain the so-called *safe-diagnoser*, denoted by $G^{\mathrm{diag,s}}$, in which some states are labeled as *bad states* since they are reachable by executing strings in $\mathscr{K}_f$. As explained in [68], the safe-diagnoser can be used to test (off-line) the property of safe diagnosability; we recall the following theorem.

Theorem 5.2 (see [68] for details.) *Consider a diagnosable language L. L is safe diagnosable with respect to projection P_{o}, fault event f, and forbidden language $\mathscr{K}_f$ if and only if in the safe-diagnoser $G^{\mathrm{diag,s}}$ built from any generator of L:*

1. *There does not exist a state q that is F-uncertain with a component of the form (x, ℓ) such that $f \in \ell$ and x is a bad state;*

2. *There does not exist a pair of states q, q' such that: (i) q is F-certain with a component of the form (x, ℓ) such that $f \in \ell$ and x is a bad state; (ii) q' is F-uncertain; and (iii) q is reachable from q' through an event $e \in E_o$.*

We have argued previously that safe diagnosability is a first necessary step in order to achieve fault tolerant supervision of DES. If the system is safe diagnosable, reconfiguration actions should be forced upon the detection of faults prior to the execution of unsafe behavior, thereby achieving the objective of fault tolerant supervision. The first step to reconfigure the system is to disable the nominal supervisor and prevent the system from executing a forbidden substring. For this purpose, it is useful to introduce the following property.

Definition 5.3 [Safe Controllable DES] *A prefix-closed language L that is live and does not contain loops of unobservable events is said to be safe controllable with respect to the projection P_{o}, fault event f, and forbidden language $\mathscr{K}_f$ if the following conditions hold:*

1. *Safe diagnosability condition: L is safe diagnosable with respect to P_{o}, f, and $\mathscr{K}_f$;*

2. *Safe controllability condition: consider any string $s \in L$ such that $f \in s$ and $s = v\sigma$ with $\sigma \in E_o$. Suppose that $\mathcal{D}$ does not hold for v while it holds for s. Then $(\forall t \in L/s)$ such that $t = u\xi$ with $\xi \in \Phi$, $\exists z \in E_c$ such that $z \in u$.*

In words, a language is safe controllable if for any string that contains a fault and a forbidden substring, there exists (i) an observable event that assures the detection of the fault before the system executes the forbidden substring and (ii) a controllable event after the observable event but before the forbidden substring. In this way, after the detection of the fault, it is always possible to disable the controllable event and avoid unsafe behavior.

Consider an automaton G generating language L and assume that L is safe diagnosable with respect to the projection P_{o}, fault event f, and forbidden language $\mathscr{K}_f$. Denote with $\mathcal{FC}$ the set of *first-entered certain states* in the safe-diagnoser $G^{\mathrm{diag,s}}$ built from G; namely, $\mathcal{FC}$ is the set of all safe-diagnoser states q such that q is F-certain and there exists a safe-diagnoser state q' which is F-uncertain and such that q is reachable from q' through an event $\sigma_{\mathrm{o}} \in E_o$. The set $\mathcal{FC}$ contains a finite number of elements:

$$\mathcal{FC} = \{q_{\mathrm{i}}\}\,,\ (i = 1 \ldots m)\,. \tag{5.7}$$

For any $q_{\mathrm{i}} = \{(x_{\mathrm{j}}, F); (x_{\mathrm{k}}, F) \ldots, (x_{\mathrm{l}}, F)\} \in \mathcal{FC}$ $(i = 1 \ldots m)$ we build a new post-fault uncontrolled model, $G_{\mathrm{i}}^{\mathrm{deg}}$, by taking the accessible part of $G^{\mathrm{n+f}}$ from all the distinct states $x_{\mathrm{j}}, x_{\mathrm{k}} \ldots x_{\mathrm{l}}$ of $G^{\mathrm{n+f}}$ that appear in the i-th safe-diagnoser state; see Fig. 5.31. To make the model deterministic, we add a new initial state $x_{0,\mathrm{i}}$ and connect it with new events called "$\mathrm{init_j}$, $\mathrm{init_k}$, $\mathrm{init_l}$" to the distinct states of $G^{\mathrm{n+f}}$ that appear in the safe-diagnoser state q_{i}. The index of init is used to make these events distinct. Note that init is uncontrollable and unobservable. In practice, this accessible part will be within the faulty part of $G^{\mathrm{n+f}}$, since the occurrence of the fault has forced the system outside its original nominal behavior G^{nom}.

Using the above terminology, we can now present a procedure to test the property of safe controllability.

Proposition 5.1 *Consider automaton G generating language L and assume that L is safe diagnosable with respect to the projection P_{o}, fault event f, and forbidden language $\mathscr{K}_f$. Consider the set $\mathcal{FC}$ of first-entered certain states in the safe-diagnoser $G^{\mathrm{diag,s}}$ built from G. Language L is safe controllable if and only if $\forall q_{\mathrm{i}} \in \mathcal{FC}$, language $\{\varepsilon\}^{\downarrow C}$, computed with respect to the post-fault uncontrolled model $G_{\mathrm{i}}^{\mathrm{deg}}$, does not contain any element of Φ as a substring.*

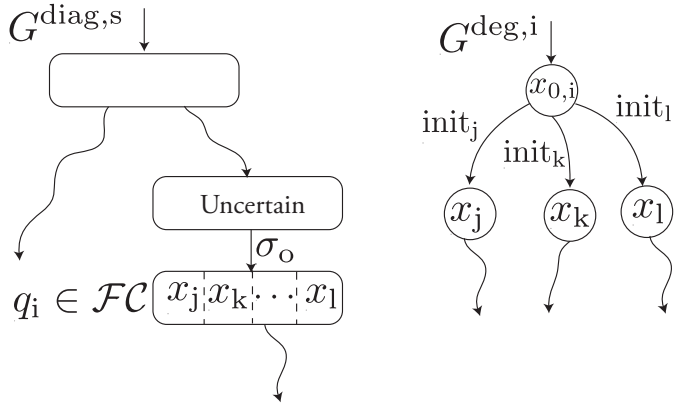


Figure 5.31: Post-fault uncontrolled model.

Proof. By construction, $\{\varepsilon\}^{\downarrow C}$ computed with respect to $G_{\mathrm{i}}^{\mathrm{deg}}$ contains all the shortest continuations in $\mathcal{L}(G^{\mathrm{n+f}})$ after detecting the occurrence of fault f (in the i-th state in $\mathcal{FC}$), in which the system can be controlled, i.e., the possible evolutions in $G_{\mathrm{i}}^{\mathrm{deg}}$ after disabling all the events that can be feasibly disabled. Suppose that some string in this controllable language contains an element of Φ as a substring; this means that there is no way to prevent the system from executing a forbidden sequence after the detection of fault f in the i-th state in $\mathcal{FC}$. This is a violation of safe controllability.
Next, suppose that the safe controllability condition does not hold for language $\mathcal{L}(G^{\mathrm{n+f}})$, i.e., for at least one string $s \in \mathcal{L}(G^{\mathrm{n+f}})$ such that $f \in s$ and $s = v\sigma$ with $\sigma \in E_o$ and such that $\mathcal{D}$ does not hold for v while it holds for s, there exists at least one continuation $t \in \mathcal{L}(G^{\mathrm{n+f}})/s$ such that $t = u\xi$ with $\xi \in \Phi$ for which $\nexists z \in E_c$ such that $z \in u$. Language $\{\varepsilon\}^{\downarrow C}$ computed with respect to $G_{\mathrm{i}}^{\mathrm{deg}}$ contains all the concatenations of uncontrollable events feasible in $\mathcal{L}(G_{\mathrm{i}}^{\mathrm{deg}})$; since by hypothesis there does not exist any controllable event in between the detection of the fault and before executing the forbidden sequence in Φ, there exists at least a string in $\{\varepsilon\}^{\downarrow C}$ that contains an element of Φ as a substring. $\triangleleft$

Remark 5.1 *Standard techniques to remove illegal substrings from a language can be used to test Proposition; see, e.g., Section 3.3 in [17].*

5.4.4 Active fault tolerance of DES

If language $\mathcal{L}(G_{\mathrm{sup}}^{\mathrm{n+f}})$ is safe controllable then it is always possible to detect any occurrence of event f in a bounded number of observable events and without executing any forbidden action; moreover, in any continuation after the detection of fault f that contains a forbidden action in Φ, there always exists at least one controllable event z that can be disabled to prevent the system from executing unsafe actions. Entering certain state $q_{\mathrm{i}} \in \mathcal{FC}$ should therefore trigger an interrupt signal $\mathrm{INT_i}$ that disables the controllable event z. Moreover, the same interrupt signal can be used to disable the nominal supervisor S^{nom} and enable a new supervisor $S_{\mathrm{i}}^{\mathrm{deg}}$ to be designed in order to meet post-fault specifications $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$. Starting from post-fault uncontrolled model $G_{\mathrm{i}}^{\mathrm{deg}}$, a set of requirements on the controlled behavior can be designed resulting in post-fault degraded specifications $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$, which in general are sublanguages of the language marked by $G_{\mathrm{i}}^{\mathrm{deg}}$. Note that $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$ need not be prefix-closed if the requirements include the ability to

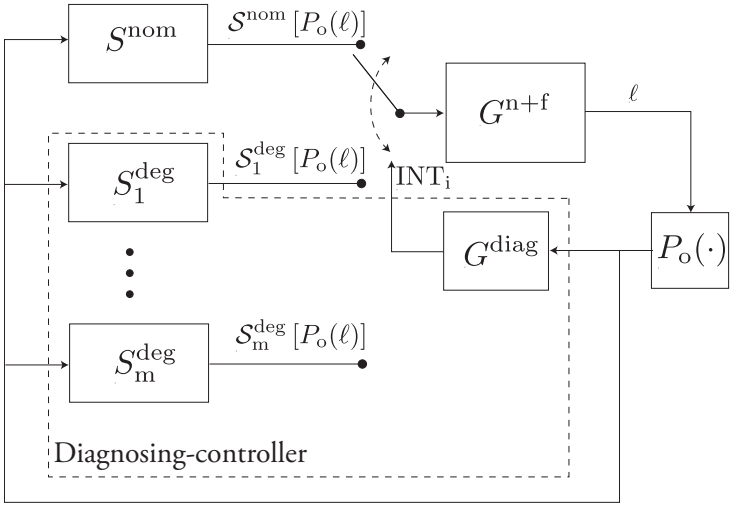


Figure 5.32: Fault tolerant supevision architecture for DES.

reach one of the so-called *recovery states* that are included and marked in $G^{\text{n+f}}$. In simpler cases, $\mathscr{K}_{\text{i}}^{\text{deg}}$ will be a prefix-closed sublanguage of the language generated by $G_{\text{i}}^{\text{deg}}$. In practice $\mathscr{K}_{\text{i}}^{\text{deg}}$ can be designed as the language generated or marked by an automaton $H_{\text{i}}^{\text{deg}}$ built by removing from $G_{\text{i}}^{\text{deg}}$ illegal states in $G^{\text{n+f}}$ and all strings that contain some undesired substring that may be specific to each $i = 1 \ldots m$. In some cases, it might be desirable to specify a minimal required behavior $\mathscr{K}_{\text{i}}^{\text{deg,min}}$ to be satisfied after the detection of the fault event f. Considering this set of degraded post-fault specifications $\mathscr{K}_{\text{i}}^{\text{deg}}$ ($i = 1 \ldots m$), we present the following definition.

Definition 5.4 [Active Fault Tolerant DES] *Language $\mathcal{L}(G^{\text{n+f}})$ is said to be active fault tolerant if for all $i = 1 \ldots m$, there exists a sublanguage of $\mathscr{K}_{\text{i}}^{\text{deg}}$ that is controllable and observable with respect to $\mathcal{L}(G_{\text{i}}^{\text{deg}})$.*

In order to test Definition 5.4 for prefix-closed specifications $\mathscr{K}_{\text{i}}^{\text{deg}}$, we can compute $\{\varepsilon\}^{\downarrow C}$ with respect to $\mathcal{L}(G_{\text{i}}^{\text{deg}})$ and test if the result is contained within $\mathscr{K}_{\text{i}}^{\text{deg}}$. Of course, this solution is likely to be impractical because it may be too restrictive. Another possibility is to compute the supremal controllable and normal sublanguage of $\mathscr{K}_{\text{i}}^{\text{deg}}$ with respect to $\mathcal{L}(G_{\text{i}}^{\text{deg}})$. This solution may also be too restrictive since the normality condition is stronger than the required observability condition. In this case, one could use existing algorithms for calculating maximal controllable and observable sublanguages of $\mathscr{K}_{\text{i}}^{\text{deg}}$; for instance, the VLP-PO algorithm presented in [40] can be used for this purpose.

For cases where $\mathscr{K}_{\text{i}}^{\text{deg}}$ is not prefix-closed, the test for active fault tolerance is more complicated, since the $\downarrow C$ operation deals with prefix-closed languages. One could still compute the supremal controllable and normal sublanguage of (marked language) $\mathscr{K}_{\text{i}}^{\text{deg}}$; however, if this approach returns the empty set, we will not know if active fault tolerance is violated, as $\mathscr{K}_{\text{i}}^{\text{deg}}$ could still possess a controllable and observable sublanguage. In this case, the recent results in [99] could be used to test the existence or not of such a language. However, a positive test may still yield a solution that is deemed too restrictive. More research is required regarding the development of algorithms for computing controllable and observable sublanguages of non-prefix-closed languages.

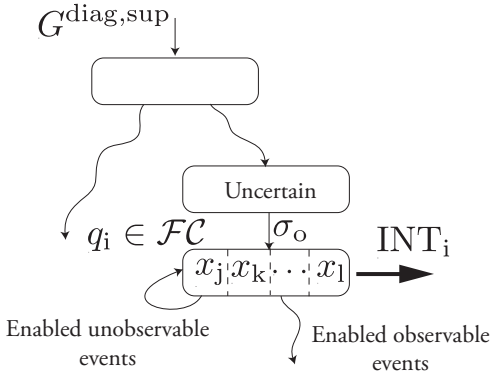


Figure 5.33: The diagnosing-controller for the example in Fig. 5.31.

In Fig. 5.32 a possible architecture for active fault tolerant control of DES is presented. During nominal functioning, the partial observation loop is closed on the nominal supervisor S^{nom} which, recording the observation $P_{\mathrm{o}}(\ell)$, issues the control action $\mathcal{S}^{\mathrm{nom}}[P_{\mathrm{o}}(\ell)]$ that encodes the enabled events after the system $G^{\mathrm{n+f}}$ executes the string ℓ. In parallel to the control loop, the diagnoser[1] G^{diag} uses the same observations to detect occurrences of the fault event f. If the system is safe diagnosable, after any occurrence of event f, the diagnoser detects the fault in a bounded number of events and before the system executes any forbidden string in Φ. When the diagnoser becomes F-certain entering state $q_{\mathrm{i}} \in \mathcal{FC}$ (mapped from the safe-diagnoser), the interrupt signal $\mathrm{INT_i}$ is issued. If the system is safe controllable, we know that it is possible to stop the evolution of $G^{\mathrm{n+f}}$ before it executes forbidden substrings in Φ. The same interrupt signal is used to switch from the nominal supervisor S^{nom} to the post-fault supervisor $S_{\mathrm{i}}^{\mathrm{deg}}$ that issues the control actions $\mathcal{S}_{\mathrm{i}}^{\mathrm{deg}}[P_{\mathrm{o}}(\ell)]$.

The existence of this supervisor is assured if the active fault tolerance property holds; in this case, the behavior of G^{n+f} can be controlled in order to satisfy the specification $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$. As depicted in Fig. 5.32, it is possible to embed both the diagnoser G^{diag} and the bank of post-fault supervisors $S_{\mathrm{i}}^{\mathrm{deg}}$ in a unique unit called the *diagnosing-controller*, whose structure is shown in Fig. 5.33.

The diagnosing-controller is an automaton built from the diagnoser G^{diag} and considering the model $G^{\mathrm{n+f}}$. If $G_{\mathrm{sup}}^{\mathrm{n+f}}$ is safe diagnosable, then after any occurrence of fault event f the diagnoser enters, in a bounded number of events, a first-entered certain state $q_{\mathrm{i}} = \{(x_{\mathrm{j}}, F); (x_{\mathrm{k}}, F) \ldots, (x_{\mathrm{l}}, F)\} \in \mathcal{FC}$ (again, mapped from the safe-diagnoser) and, after that moment, all the other reachable states in G^{diag} are certain states. When G^{diag} enters q_{i}, the signal $\mathrm{INT_i}$ is generated and used to disable the controllable event z to avoid any occurrence of forbidden substrings; moreover the same event is used to disable the nominal supervisor S^{nom}. Considering these facts, when G^{diag} enters q_{i}, we are sure that event f has occurred and the actual state in $G^{\mathrm{n+f}}$ is one of the states in the list of q_{i}, i.e. $x_{\mathrm{j}}; x_{\mathrm{k}} \ldots x_{\mathrm{l}}$. Note that if the system is safe diagnosable, none of the states $x_{\mathrm{j}}; x_{\mathrm{k}} \ldots x_{\mathrm{l}}$ will be reached by the execution of forbidden substrings (see [68]). As previously stated, $x_{\mathrm{j}}; x_{\mathrm{k}} \ldots x_{\mathrm{l}}$ can be considered as initial states for the uncontrolled i-th post-fault model of the system. Moreover, the post-fault evolution can be reconstructed considering the connectivity in $G^{\mathrm{n+f}}$ starting from states $x_{\mathrm{j}}; x_{\mathrm{k}} \ldots x_{\mathrm{l}}$. At this point,

[1]The standard diagnoser is used here as it is suffices for the present purpose. Relevant mapping of states is done from the safe-diagnoser to the diagnoser regarding the set $\mathcal{FC}$.

if $\mathcal{L}(G^{\mathrm{n+f}})$ is active fault tolerant with respect to post-fault specification $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$, it is possible to design a post-fault supervisor $S_{\mathrm{i}}^{\mathrm{deg}}$. As depicted in Fig. 5.33, the realization of the post-fault supervisor $S_{\mathrm{i}}^{\mathrm{deg}}$ can be considered as directly connected to first-entered F-certain state q_{i} in the diagnoser.

In view of the preceding discussion, we present an algorithmic procedure to build the diagnosing-controller.

Procedure to build the diagnosing-controller.

Step 1: Build the diagnoser G^{diag} from $G_{\mathrm{sup}}^{\mathrm{n+f}}$;

Step 2: For any $q_{\mathrm{i}} = \{(x_{\mathrm{j}}, F); (x_{\mathrm{k}}, F) \ldots, (x_{\mathrm{l}}, F)\} \in \mathcal{FC}$;

- **Step 2.1:** Stop the evolution of G^{diag} after q_{i} and enable signal $\mathrm{INT}_{\mathrm{i}}$ when entering q_{i};
- **Step 2.2:** Build the post-fault model $G_{\mathrm{i}}^{\mathrm{deg}}$;
- **Step 2.3:** Compose $G_{\mathrm{i}}^{\mathrm{deg}}$ with a realization $H_{\mathrm{i}}^{\mathrm{deg}}$ of specification language $\mathscr{K}_{\mathrm{i}}^{\mathrm{deg}}$. Define $R^{\mathrm{deg}} = H_{\mathrm{i}}^{\mathrm{deg}} \times G_{\mathrm{i}}^{\mathrm{deg}}$;
- **Step 2.4:** Starting from R^{deg}, build the post-fault supervisor realization $S_{\mathrm{i}}^{\mathrm{deg}}$ using techniques from supervisory control theory;
- **Step 2.5:** Overlap the initial state of $S_{\mathrm{i}}^{\mathrm{deg}}$ with state q_{i} of G^{diag};

Step 3: Call the resulting automaton $G^{\mathrm{diag,sup}}$.

We discuss the computational complexity of the above procedure. We know from [78] that the complexity of Step 1 is in the worst case exponential in the cardinality of the state set of automaton $G_{\mathrm{sup}}^{\mathrm{n+f}}$. Hence the cardinality of the set of first entered certain states $\mathcal{FC}$ is worst-case exponential as well, although this upper bound is very unlikely to be reached in practical applications. Experience with applications of the Diagnoser Approach has shown that due to the structure of real systems, their diagnosers usually have a state space whose cardinality is of the same order as that of the original system. Moreover, the cardinality of the set $\mathcal{FC}$ is expected to be much smaller than that of the state set of the diagnoser.

For any diagnoser state $q_{\mathrm{i}} = \{(x_{\mathrm{j}}, F); (x_{\mathrm{k}}, F) \ldots, (x_{\mathrm{l}}, F)\} \in \mathcal{FC}$, Steps 2.1 to 2.3 build the post-faults models using reachability analysis techniques and thereby have polynomial complexity in the state set of $G^{\mathrm{n+f}}$. Finally, the complexity of Step 2.4 depends on the technique used to design the post-fault supervisor realization $S_{\mathrm{i}}^{\mathrm{deg}}$. Recall the discussion following Definition 5.4. Supervisor synthesis in the case of partial observation has in the worst-case exponential complexity in the state space of the given initial condition (here, R^{deg}), since it involves a determinization step due the presence of unobservable events. This determinization step is needed to ensure that a deterministic realization of the control law is obtained. An alternative to complete off-line syntehsis is to adopt on-line control techniques for Step 2.4 (such as those in [40]), which typically have polynomial complexity at each observable event along a system trajectory.

5.4.5 An illustrative example

Consider the hydraulic system of Fig. 5.34 (a); the system is composed of a tank T, a pump P, a set of valves (V1, V2, and Vr), and associated pipes. The pump P is used to move fluid from the tank through the pipe and must be coordinated with the set of redundant valves. The system

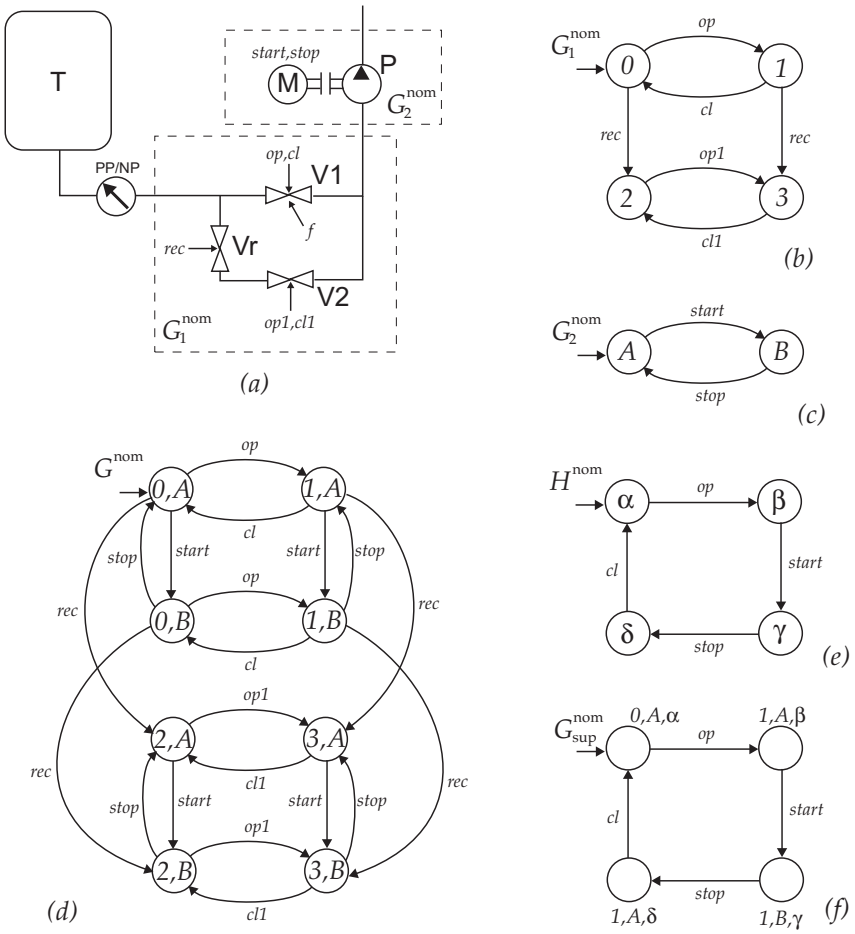


Figure 5.34: The hydraulic system example: (a) the system; (b) nominal model G_1^{nom} for the set of valves; (c) nominal pump model G_2^{nom}; (d) global nominal model G^{nom}; (e) nominal specification H^{nom}; (f) nominal supervised system $G_{\text{sup}}^{\text{nom}}$.

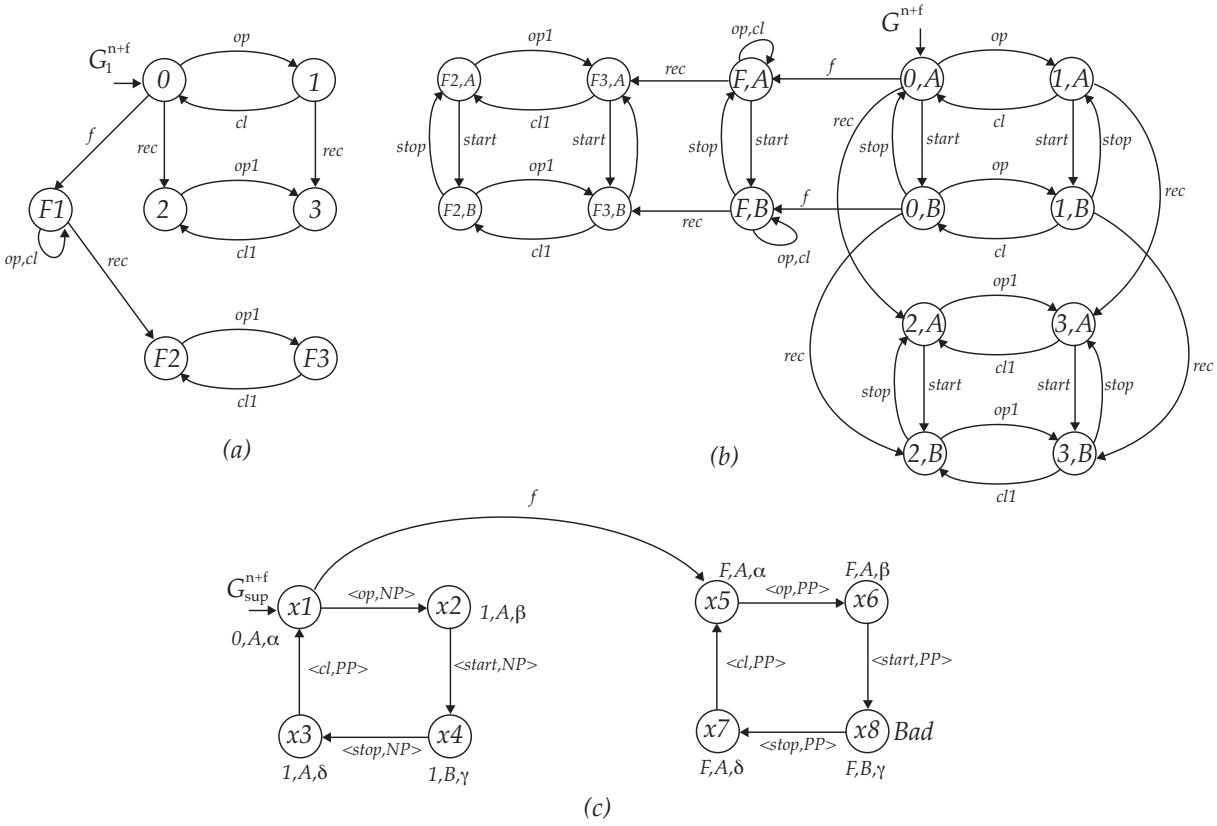


Figure 5.35: The hydraulic system example: (a) model of valves with fault f, $G_1^{\mathrm{n+f}}$; (b) complete model $G^{\mathrm{n+f}} = G_1^{\mathrm{n+f}} \| G_2^{\mathrm{nom}}$; (d) complete supervised model $G_{\mathrm{sup}}^{\mathrm{n+f}}$.

is equipped with a pressure sensor. The automaton modeling the set of valves is denoted by G_1^{nom} and is shown in Fig. 5.34 (b): events op and cl are used to open and close valve V1, events $op1$ and $cl1$ are used to open and close valve V2, event rec is used to open safety valve Vr. All these events are observable and controllable. In Fig. 5.34 (c) the model G_2^{nom} of the pump P is shown; events $start$ and $stop$, observable and controllable, are used to switch on and off the pump, respectively. In Fig. 5.34 (d) the nominal model of the system $G^{\mathrm{nom}} = G_1^{\mathrm{nom}} \| G_2^{\mathrm{nom}}$ is depicted; this model must be controlled according to the specification defined by automaton H^{nom} shown in Fig. 5.34 (e). It is easy to see that $\mathcal{L}(H^{\mathrm{nom}})$ is controllable and observable with respect to $\mathcal{L}(G^{\mathrm{nom}})$ and the supervised behavior $G_{\mathrm{sup}}^{\mathrm{nom}}$ is drawn in Fig. 5.34 (f).

Due to a malfunction, valve V1 may get stuck closed; this fact is modeled using unobservable and uncontrollable event f and the model of the valves $G_1^{\mathrm{n+f}}$ is depicted in Fig. 5.35 (a). According to this refined model, the automaton $G^{\mathrm{n+f}} = G_1^{\mathrm{n+f}} \| G_2^{\mathrm{nom}}$ modeling the uncontrolled system is depicted in Fig. 5.35 (b). In Fig. 5.35 (c) the effect of the nominal supervision policy on $G^{\mathrm{n+f}}$ is denoted by $G_{\mathrm{sup}}^{\mathrm{n+f}}$. In Fig. 5.35 (c) pressure sensor readings are attached to events; note that if valves are closed, the sensor reads an over pressure in the pipe (PP), while, if valves are open, the sensor reads no over-pressure in the pipe (NP). The situation in which the pump is working with closed valves has to be avoided because it is unsafe: $\Phi = \{start\}$. Note that this situation is feasible in $G_{\mathrm{sup}}^{\mathrm{n+f}}$ where state $x8$ is labeled as bad (see [68]).

In Fig. 5.36 (a), the safe-diagnoser of $G_{\mathrm{sup}}^{\mathrm{n+f}}$ is shown. Since the bad state $x8$ is not present

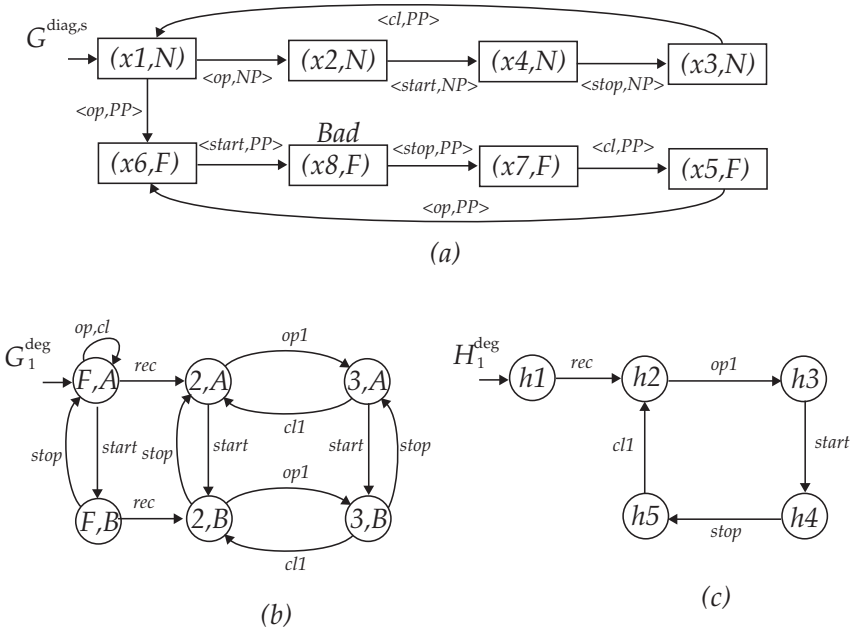


Figure 5.36: The hydraulic system example: (a) safe diagnoser $G^{\mathrm{diag,s}}$; (b) post-fault model G_1^{deg}; (c) post-fault specification H_1^{deg}.

in any of the uncertain states or in a first-entered certain state, the system is safe diagnosable; in this case the set of first-entered certain states is $\mathcal{FC} = \{(x6, F)\}$. Figure 5.36 (b) shows the feasible evolution G_1^{deg} after detection; here, no new initial state and init event is needed since G_1^{deg} is deterministic.

It is easy to prove that $\{\varepsilon\}^{\downarrow C}$ computed with respect to $\mathcal{L}(G_1^{\mathrm{deg}})$ is equal to $\{\varepsilon\}$, therefore $G_{\mathrm{sup}}^{\mathrm{n+f}}$ turns out to be safe controllable. Starting from G_1^{deg}, the post-fault specification generated by automaton H_1^{deg} in Fig. 5.36 (c) can be designed; note the use of the event rec by which the safety valve is opened letting the system continue performing its operation using the redundant valve V2. Since this specification turns out to be controllable and observable with respect to G_1^{deg}, the system is active fault tolerant.

Finally, the diagnosing-controller $G^{\mathrm{diag,sup}}$ for this example is shown in Fig. 5.37. Since in this example the safe-diagnoser and the standard diagnoser are the same automaton, the diagnosing-controller can be obtained from the safe-diagnoser in Fig. 5.36 (a) by stopping its construction at the first-entered certain state $(x6, F)$ and suitably overlapping to this state the post-fault supervisor realization S_1^{deg}.

It is important to stress how, entering state $(x6, F)$, signal INT is enabled, by which the nominal supervisor is disabled and the forbidden action $start$ is temporarily disabled.

5.4.6 Conclusions on active fault tolerant control using online diagnostic

The main contributions of this work can be summarized as follows.

1. It has investigated an active approach to FTC of DES that makes use of a multiple-supervisor architecture to actively counteract the effect of faults. The control algorithm

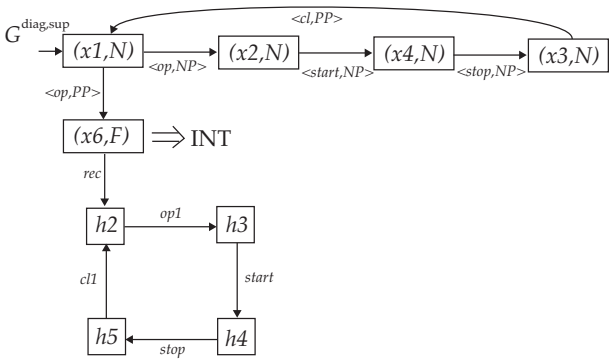


Figure 5.37: The hydraulic system example: the diagnosing-controller $G^{\mathrm{diag,sup}}$.

employs online diagnosics to actively react to the detection of a malfunctioning component in order to eventually meet degraded control specifications.

2. It has evaluated the effect of the diagnostics algorithm on the FTC architecture, based on the idea that "fast"fault-detection is needed to promptly react and reconfigure the system before it executes some unsafe action. To this aim, starting from an appropriate model of the system, we have shown how the notion of safe diagnosability is a necessary step in order to achieve fault tolerant supervision of DES.

3. It has introduced the new notions of "safe controllability"and "active fault tolerant system"to characterize the conditions that must be satisfied when solving the FTC problem using the active approach. Computational tests for these properties were presented.

4. It has proposed a procedure to design an automaton called "diagnosing-controller"which, embedding the diagnoser as well as the set of reconfigured controllers, is able, if the system satisfies the properties introduced, to solve the fault tolerant supervision problem.

Future research in this area could include the introduction of temporal behavior in the system model and the consideration of a decentralized/distributed FTC architecture. This last point is of particular interest as it paves the way to the definition of meaningful reconfiguration strategies in which small faults are counteracted at the node level, whereas severe faults are managed at a higher level involving a set of nodes.

Conclusions and future works

The main aim of this thesis, conclusive work of a three years research period focused on industrial automation software architectures, is the development of a design pattern for control logic in industrial automated systems. As explained in this work, in last years the industrial automation processes have acquired an ever increasing degree of automation, this trend derives by the higher requirement of systems in term of quality, productivity, efficiency. This increasing make complex the design of an industrial automated systems and in particular of its control system. This works starts with a survey chapter on automated industrial a systems, with a particular mention to automated manufacturing systems, which illustrates the main characteristic of this system and some definition. Starting from this, and analyzing the state of the art of software engineering in industrial automation and the usually specification in an AMS we arrived to define an architecture based on the new concept of the *Generalized Device* (GA). The main concept is the separation between the policy as a sequence of actions, and the actuation mechanisms which perform the actions. This entity is defined in order to (i) encapsulate actuation mechanisms separating them from high level control policies in order to hierarchically manage the plant, (ii) support hardware virtualization, component interoperability and reusability and (iii) make easier the design of diagnostics, reconfiguration and quality check functionalities exploiting a distributed and hierarchical approach. Inside the work some examples show how this architecture has characteristic of modularity and composability. A lack in this approach is the low reusability of the software, because is dependent from the particular hardware of the AMS. Starting from this point it was take under consideration this question "Do different field devices really need different control logic ?"
In the GA approach an actuation mechanism was a set of an actuators and sensors to perform an action. In AMS it is possible to find different devices from different fields (electric, pneumatic, etc.) but analyzing the kind of signals that sensors and actuators need to exchange to perform an action, we can define a new entity called *Generalize Device*. In this way GD is "component"independent from the hardware, and so it is reusability. The main characteristic of this architecture is the standardization of set of devices and a division of the diagnostic in two level: a low level and a high level diagnostic. The low level diagnostic is embedded inside the GD component, in this level are presented algorithms of fault detection on sensors and actuators faults (i.e. stuck low, stuck high) this kind of fault are independent from hardware and the specific application of the AMS. High level faults means faults which are dependent from the application and to detect this faults it need use information generates from two or more devices.
AMS are vulnerable systems so the diagnosis problem is an important topic. The faults isolation in a generic dynamical system consists in the design of an elaboration unit that, appropri-

ately processing the inputs and outputs of the dynamical system, is also capable of detecting incipient faults on the plant devices, reconfiguring the control system so as to guarantee satisfactory performance. In the Gd component are embedded fault detection and fault isolation algorithms but to guarantee the effectiveness of the diagnosis system, it should also keep into account an appropriate plant model, that describes its behaviour starting from the knowledge of some inputs and outputs. In the last part of this work, it was presented a general and versatile approach for building structured formal models in order to facilitate their control and diagnosis using techniques from discrete events system theory. For this purpose, it was presented a methodology for building in a modular manner the complete model of a complex automated system starting from individual components and their physical coupling. Using this approach it is possible to have the faulty model of the entire system and using this have a formal proof that the fault detection algorithm are correct. Finally, also the topic of the fault tolerant control was explained.
Future development of this work will be the study and the implementation of tools which are able to support the fulfillment of concepts of the Generalized Actuator and Generalized Device, in particular an important point should be a definition of formal methods for control code generation to bridge the obtained results with the field of industrial informatics in order to obtain (semi)automatic code generation with respect to standard programming languages.
Another very interesting research problem is the definition of a modeling framework able to describe functionalities as well as the effect of their allocation and implementation over physical resources. This modeling tool will be used to model architectures and components for computer-based distributed control systems with a clear distinction between functional and implementative layer. This architecture should be modular and hierarchical in order to represent, at different abstraction levels, the distributed nature of the system.
All this effort will go in the direction to the definition of formal methods for the design and verification of the functional architecture; this point is devoted to the study of new supervisory control architectures that, exploiting the distributed nature of the systems (decentralized control, modular control etc.) and considering the effect of the integration with the implementation layer, allow the compliance of key specifications such as safety and fault tolerance.

Appendixes

Appendix A

Introduction to discrete event systems theory

In this appendix some notions about discrete event dynamical systems (DEDS) are reported, including basic definitions about discrete event systems (DES), theory of automata and languages and some important results about feedback supervision. The interested reader is referred for a more complete and formal treatment of these topics to [17], [98] and references therein.

A.1 Discrete event systems

A *Discrete event system* (DES) is a dynamical system in which the state space is naturally described by a discrete set, and the state transitions are only observed at discrete points in time. We associate the state transitions with "events"; an event can be identified with a specific action taken. This action can be spontaneous (dictated by nature) or it may be the result of several conditions which are suddenly met. The symbol e denotes an event; considering a system affected by different types of events, we will assume we can define an *event set* E, whose elements are all these events.
Discrete event systems satisfy the following two properties:

1. The state space is a *discrete set*.
2. The state transition mechanism is *event drive*.

Definition A.1 *A discrete event system is a discrete-state, event-driven system, i.e. its state depends entirely on the occurrence of asynchronous discrete events over time.*

With this in mind, the behavior of a DES can be described in terms of event sequences of the form $e_1, e_2, \cdots, e_n$. A more formal way to study the logical behavior of DES is based on the theories of languages and automata.

The starting point is the fact that any DES has an underlying event set E associated with it. The set E is thought of as the *alphabet* of a language and event sequences are thought of as *strings* (words) in that language. A string consisting of no events is called *empty string* and is denoted by ϵ. The length of a string s, denoted with $|s|$, is the number of events contained in it, counting multiple occurrences of the same event.

Definition A.2 *A* language *defined over a set E is a set of finite-length strings from events in E.*

The key operation involved in building strings and thus languages from a set of events E is *concatenation*: the string abb is the concatenation of the string ab and the event b. The empty string ϵ is the identity element of concatenation. Let $E^\star$ denote the set of all finite strings of elements of E, including the empty string ϵ; the $(\cdot)^\star$ operation is called *Kleene-closure*. A language over an event set E is a subset of $E^\star$.
If $tuv = s$, the following nomenclature can be defined:

- t is called a *prefix* of s,
- u is called a *substring* of s,
- v is called a *suffix* of s.

Observe that both ϵ and s are prefixes, substrings and suffixes of s.

A.2 Operations on Languages

The usual set operations, such as union, intersection, difference and complement with respect to $E^\star$ are applicable to languages, since languages are sets. In addition it is possible to introduce the following operations:

- *Concatenation*: Let $L_a, L_b \subseteq E^\star$, then

  $$L_a L_b := \{s \in E^\star : (s = s_a s_b) \text{ and } (s_a \in L_a) \text{ and } (s_b \in L_b)\} \ .$$

  A string is in $L_a L_b$ if it can be written as the concatenation of a string in L_a with a string in L_b.

- *Prefix-closure*: Let $L \subseteq E^\star$, then

  $$\overline{L} := \{s \in E^\star : \exists t \in E^\star (st \in L)\} \ .$$

  The prefix closure of L is the language $\overline{L}$ consisting of all the prefixes of all the strings in L. In general $L \subseteq \overline{L}$.

- *Kleene-closure*: Let $L \subseteq E^\star$, then:

  $$L^\star := \{\epsilon\} \cup L \cup LL \cup LLL \cup \cdots \ .$$

  An element of $L^\star$ is formed by the concatenation of a finite number of elements of L.

A.3 Representation of languages: automata

An automaton is a device that is capable of representing a language according to well-defined rules.

Definition A.3 *A* deterministic automaton*, denoted by G, is a six-tuple*

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where

- *X is the set of states*
- *E is the finite set of events associated with transitions in G*
- *$f : X \times E \to X$ is the transition function: $f(x, e) = y$ means that there is a transition labelled by event e from state x to state y; in general f is a partial function on its domain*
- *$\Gamma : X \to 2^E$ is the active event function: $\Gamma(x)$ is the set of all the events e for which $f(x, e)$is defined*
- *x_0 is the initial state*
- *$X_M \subseteq X$ is the set of marked states.*

The automaton is said to be deterministic because f is a function over $X \times E$. In contrast the transition function of a nondeterministic automaton is defined by means of a relation over $X \times E \times X$[1].
An automaton generates a language defined in the following way.

Definition A.4 *The* language generated *by $G = (X, E, f, \Gamma, x_0, X_m)$ is*

$$\mathcal{L}(G) := \{s \in E^\star : f(x_0, s) \text{ is defined}\} \ . \tag{A.1}$$

The language marked *by $G = (X, E, f, \Gamma, x_0, X_m)$ is*

$$\mathcal{L}_m(G) := \{s \in E^\star : f(x_0, s) \in X_m\} \ . \tag{A.2}$$

In other words a string s is in $\mathcal{L}(G)$ if and only if it corresponds to an admissible path in the state transition diagram. Note that in the above definitions we work with an extension of the transition function defined over $X \times E^\star$ as:

$$\begin{array}{lcl} f(x, \epsilon) & := & x \\ f(x, se) & := & f(f(x, s), e) \text{ for } s \in E^\star \text{ and } e \in E \ . \end{array}$$

Two automata are said to be *equivalent* if they generate and mark the same languages, i.e.

Definition A.5 *Automata G_1 and G_2 are said to be equivalent if*

$$\mathcal{L}(G_1) = \mathcal{L}(G_2) \quad \text{and} \quad \mathcal{L}_m(G_1) = \mathcal{L}_m(G_2) \ .$$

[1] Or equivalently a function from $X \times E$ to 2^X.

In general the following property holds:

$$\mathcal{L}_m(G) \subseteq \overline{\mathcal{L}(G)} \subseteq \mathcal{L}(G) \ .$$

It can happen that an automaton G could reach a state x where $\Gamma(x) = \emptyset$, but $x \notin X_m$. This is said a *deadlock*, because no further event can be executed. If deadlock happens, then necessarily $\overline{\mathcal{L}_m(G)}$ will be a proper subset of $\mathcal{L}(G)$, since any string in $\mathcal{L}(G)$ that ends at state x cannot be a prefix of a string in $\mathcal{L}_m(G)$.

Consider now the case when there is set of unmarked states in G that forms a strongly connected component, but with no transitions going out of the set. If the system enters this set, then we get a so-called *livelock*. If livelock is possible then again $\overline{\mathcal{L}_m(G)}$ will be a proper subset of $\mathcal{L}(G)$.

Definition A.6 *An automaton G is said to be blocking if*

$$\overline{\mathcal{L}_m(G)} \subset \mathcal{L}(G)$$

and nonblocking when

$$\overline{\mathcal{L}_m(G)} = \mathcal{L}(G) \ .$$

In other words if an automaton is blocking either deadlock and/or livelock can happen.

Suppose now that an event e at state x may cause transitions to more than one state. In this case $f(x, e)$ is represented by a set of states. In addition we may want that to allow the label ϵ in the state transition diagram, i.e. we allow transitions between distinct states to have the empty string as label[2]. These two changes lead to the definition of a *nondeterministic automaton*.

Definition A.7 *A* nondeterministic automaton*, denoted by G_{nd}, is a six-tuple*

$$G_{nd} = (X, E \cup \{\epsilon\}, f_{nd}, \Gamma, x_0, X_m)$$

where

- *f_{nd} is a function $f_{nd} : X \times E \cup \{\epsilon\} \to 2^X$, that is $f_{nd}(x, e) \subseteq X$ whenever it is defined*
- *The initial state may be itself a set of states: $x_0 \subseteq X$.*

A.3.1 Operations on automata

- *Accessible part*: From the definition of $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$, we can delete from G all states that are not accessible or reachable from x_0 by some string in $\mathcal{L}(G)$ without affecting the languages generated and marked by G. When we delete a state we also delete all the transitions that are attached to that state. We will denote this operation by $Ac(G)$ (taking the accessible part).

$$\begin{array}{lcl} Ac(G) & := & (X_{ac}, E, f_{ac}, x_0, X_{ac,m}) \\ X_{ac} & := & \{x \in X : \exists s \in E^\star (f(x_0, s) = x)\} \\ X_{ac,m} & := & X_m \cap X_{ac} \\ f_{ac} & := & f|_{X_{ac} \times E \to X_{ac}} \ . \end{array}$$

[2]These transitions may represent events that cause a change in the internal state but are not observable by an outside observer.

- *Coaccessible part*: A state x of G is said to be coaccessible to X_m if there is a string in $\mathcal{L}_m(G)$ that goes through x; i.e. there is a path in the state transition diagram of G from state x to a marked state. We denote the operation of deleting all the states of G that are not coaccessible by $CoAc(G)$:

$$\begin{array}{lcl} CoAc(G) & := & (X_{coac}, E, f_{coac}, x_{0,coac}, X_m) \\ X_{coac} & := & \{x \in X : \exists s \in E^\star (f(x,s) \in X_m)\} \\ x_{0,coac} & := & \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ \text{undefined} & \text{otherwise} \end{cases} \\ f_{coac} & := & f|_{X_{coac} \times E \to X_{coac}} \ . \end{array}$$

  The $CoAc$ operation may shrink $\mathcal{L}(G)$, but does not affect $\mathcal{L}_m(G)$.

- *Trim operation*: An automaton that is both accessible and coaccessible is said to be trim. We define the $Trim$ operation as:

$$Trim(G) = CoAc\,[Ac(G)] = Ac\,[CoAc(G)]\,.$$

- *Complement*: Consider a trim automaton $G = (X, E, f, \Gamma, x_0, X_m)$ that marks the language $L \subseteq E^\star$, we can define a complement automaton G^{comp} that will mark the language $E^\star \backslash L$.

- *Product*: consider the two automata

$$G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1}) \quad \text{and} \quad G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$$

  the product of G_1 and G_2 is the automaton

$$G_1 \times G_2 = Ac\,(X_1 \times X_2, E_1 \cap E_2, f, \Gamma_{1\times 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

  where

$$f((x_1,x_2),e) = \begin{cases} (f_1(x_1,e), f_2(x_2,e)) & \text{if } \ e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

  and thus

$$\Gamma_{1\times 2}(x_1,x_2) = \Gamma_1(x_1) \times \Gamma_2(x_2)\ .$$

  This means that in the product the transitions of the two automata must always be synchronized on common events (in $E_1 \cap E_2$). It is easy to verify that:

$$\mathcal{L}(G_1 \times G_2) = \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \qquad \mathcal{L}_m(G_1 \times G_2) = \mathcal{L}_m(G_1) \cap \mathcal{L}_m(G_2)$$

- *Parallel composition*: consider the two automata

$$G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1}) \quad \text{and} \quad G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$$

  the parallel composition of G_1 and G_2 is the automaton

$$G_1 \parallel G_2 = Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1\|2}, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

  where

$$f((x_1,x_2),e) = \begin{cases} (f_1(x_1,e), f_2(x_2,e)) & \text{if } \ e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1,e), x_2) & \text{if } e \in \Gamma_1(x_1) \backslash E_2 \\ (x_1, f_2(x_2,e)) & \text{if } e \in \Gamma_2(x_2) \backslash E_1 \\ \text{undefined} & \text{otherwise .} \end{cases}$$

In the parallel composition a common event can only be executed if the two automata both execute it simultaneously. The two automata are synchronized on common events. To characterized the language generated, we define the *projection*

$$P_i : (E_1 \cup E_2)^\star \to E_i^\star \quad \text{for } i = 1, 2$$

as follows:

$$\begin{aligned} P_i(\varepsilon) &= \epsilon \\ P_i(e) &= \begin{cases} e & \text{if } e \in E_i \\ \epsilon & \text{if } e \notin E_i \end{cases} \\ P_i(se) &= P_i(s)P_i(e) \text{ for } s \in (E_1 \cup E_2)^\star, e \in (E_1 \cup E_2). \end{aligned}$$

In other words given two event sets where one is a subset of the other, this kind of projection (called natural projection) erases events in a string formed from the larger set, that do not belong to the smaller one. We can also introduce the corresponding inverse maps (*inverse projection*)

$$P_i^{-1} : E_i^\star \to 2^{(E_1 \cup E_2)^\star}$$

defined as:

$$P_i^{-1} = \{s \in (E_1 \cup E_2)^\star : P_i(s) = t\} .$$

In other words given a string in the smaller event set, the inverse projection returns the set of all strings in the larger event set that project to the given string. The projections and their inverses are extended to languages, simply by applying them to all the strings in the language. Note that

$$P_i \left[P_i^{-1}(L)\right] = L$$

but in general

$$L \subseteq P_i^{-1} \left[P_i(L)\right] .$$

Returning to the parallel composition between automata, it easy now to prove that

$$\mathcal{L}(G_1 \parallel G_2) = P_1^{-1} \left[\mathcal{L}(G_1)\right] \cap P_2^{-1} \left[\mathcal{L}(G_2)\right] \qquad \mathcal{L}_m(G_1 \parallel G_2) = P_1^{-1} \left[\mathcal{L}_m(G_1)\right] \cap P_2^{-1} \left[\mathcal{L}_m(G_2)\right] .$$

A.3.2 Observer automata

It is always possible transform a nondeterministic automaton G_{nd} into an equivalent deterministic one. We will call the resulting equivalent deterministic automaton the *observer* G_{obs} corresponding to the nondeterministic automaton. The procedure to build the automaton can be found in [17]. It is important just to recall the properties of the observer:

1. G_{obs} is a deterministic automaton.
2. $\mathcal{L}(G_{obs}) = \mathcal{L}(G_{nd})$
3. $\mathcal{L}_m(G_{obs}) = \mathcal{L}_m(G_{nd})$

We motivated previously the use of ϵ-transitions in a nondeterministic automaton as events that occur in the system modelled by the automaton, but cannot be observed from outside. Those events are considered as *unobservable events*, in other words the event set is partitioned into two disjoints parts:

$$E = E_o \cup E_{uo}$$

where E_o is the set of observable events and E_{uo} is the set of unobservable events. Treating unobservable events as ϵ-transitions and building the observer corresponding to the nondeterministic automaton obtained, it is easy to prove that the observer satisfy the following properties:

- $\mathcal{L}(G_{obs}) = P\left[\mathcal{L}(G)\right]$
- $\mathcal{L}_m(G_{obs}) = P\left[\mathcal{L}_m(G)\right]$
- The state of G_{obs} that is reached after a string $t \in P\left[\mathcal{L}(G)\right]$ will contain all the states of G that can be reached after any strings in

$$P^{-1}(t) \cap \mathcal{L}(G) \ .$$

Where P denotes the natural projection from E to E_o defined as follows:

$$\begin{array}{lcl} P(\epsilon) & = & \epsilon \\ P(e) & = & \left\{ \begin{array}{ll} e & \text{if } e \in E_o \\ \epsilon & \text{if } e \notin E_o \end{array} \right. \\ P(se) & = & P(s)P(e) \text{ for } s \in E^\star, e \in E. \end{array}$$

In other words, the state of G_{obs} is the union of all the states of G that are consistent with the observable events that have occurred so far. In this sense the state of G_{obs} is an estimate of the current state of G.

A.4 Regular languages

Any language can be marked by an automaton: simply build the automaton as a possibly infinite tree whose root is the initial state and where the nodes at layer n of the tree are entered bye the strings of length n or the prefixes of length n of the longer strings. The state space is the set of nodes of the tree and a state is marked if and only if the string that reaches it from the root is an element of the language. Such tree automaton will have an infinite state space if the cardinality of the language is infinite. Of course there exist infinite languages that can be represented by finite-state automaton[3], but there exist also infinite languages that cannot be represented by finite-state automata. A classical example is the language $L = \{a^n b^n \ : \ n \geq 0\}$; see [17] for more details.

Definition A.8 *A language is said to be regular if it can be marked by a finite-state automaton. We will denote the class of regular languages by $\mathcal{R}$.*

It is easy to prove the following theorems:

Theorem A.1 *The class of languages representable by nondeterministic finite-state automata is exactly the same as the class of languages representable by deterministic finite-state automata: $\mathcal{R}$.*

Theorem A.2 *If L_1 and L_2 are in $\mathcal{R}$, then the following languages are also in $\mathcal{R}$:*

1. $\overline{L_1}$
2. $L_1^\star$

[3]The simplest case is the language $L = E^\star$ that can be represented by a single state automaton.

3. $L^c := E^\star \setminus L_1$
4. $L_1 \cup L_2$
5. $L_1 \cap L_2$
6. $L_1 L_2$

Theorem A.3 *A language is regular if and only if it can be represented by a regular expression i.e. by means of the operations of kleene-closure, union and concatenation.*

A.5 Supervisory control

The situation considered in this section is that of a given DES whose behavior must be modified by feedback control in order to achieve a given set of specifications. Consider an automaton G that models the uncontrolled behavior of the DES; this behavior is not satisfactory and must be modified by control in the sense that its behavior must be restricted to a subset of $\mathcal{L}(G)$. In this framework, we consider sublanguages of $\mathcal{L}(G)$ that represent the *legal behavior* for the controlled system. In this paradigm, the supervisor S observes some of all the events that G executes and tells G which events in its current active set are allowed next. In other words, S has the capability of disabling some feasible events of G, exerting in this way a feedback control action on G.

Consider a DES modelled by the language generated L and the language marked L_m. L and L_m are defined over the event set E. Consider the case of a prefix-closed L. These two languages are generated and marked by an automaton

$$G = (X, E, f, \Gamma, x_0, X_m) \ .$$

We want to design a supervisor S that interacts with G in a feedback manner as explained previously. Let E be partitioned into two disjoint subsets:

$$E = E_c \cup E_{uc}$$

where

- E_c is the set of *controllable events*, i.e. those events that can be prevented from happening (disabled) by supervisor S;
- E_{uc} is the set of *uncontrollable events*, i.e. those events that cannot be prevented from happening by supervisor S.

Assume for the moment that all the events in E executed by G are observed by S. A supervisor S is a function

$$S : \mathcal{L}(G) \to 2^E$$

such that for each $s \in \mathcal{L}(G)$,

$$S(s) \cap \Gamma(f(x_0, s))$$

is the set of *enabled events* that G can execute at its current state $f(x_0, s)$. In view of this we will say that supervisor S is *admissible* if for all $s \in \mathcal{L}(G)$

$$E_{uc} \cap \Gamma(f(x_0, s)) \subseteq S(s)$$

i.e. S is not allowed to ever disable a feasible uncontrollable event.

Definition A.9 *The language generated by S/G is defined recursively as follows*

1. $\epsilon \in \mathcal{L}(S/G)$
2. $[(s \in \mathcal{L}(S/G))$ *and* $(s\sigma \in \mathcal{L}(G))$ *and* $(\sigma \in S(s))] \iff [(s\sigma \in \mathcal{L}(S/G))]$.

The language marked by S/G is defined as follows:

$$\mathcal{L}_m(S/G) := \mathcal{L}(S/G) \cap \mathcal{L}_m(G).$$

Definition A.10 *The DES S/G is blocking if*

$$\mathcal{L}(S/G) \neq \overline{\mathcal{L}_m(S/G)}$$

and nonblocking if

$$\mathcal{L}(S/G) = \overline{\mathcal{L}_m(S/G)}.$$

Consider now the situation where the supervisor does not observe all the events that G executes, i.e. the event set E is partitioned into two disjoint subsets:

$$E = E_o \cup E_{uo}$$

where

- E_o is the set of *observable events*, i.e. those events that can be seen by supervisor S;
- E_{uo} is the set of *unobservable events*, i.e. those events that cannot be seen by supervisor S.

In this case the feedback loop includes a natural projection P between G and the supervisor S in the sense that the supervisor cannot distinguish between two strings s_1 and s_2 that have the same projection and will issue the same control action: $S_P[P(s_1)] = S_P[P(s_2)]$. We define a partial-observation supervisor a function

$$S_P : P[\mathcal{L}(G)] \to 2^E.$$

This means that the control action can change only after the occurrence of an observable events, i.e. when $P(s)$ changes.

Let us take the string $t = t'\sigma$ (with $\sigma \in E_o$). $S_P(t)$ is the control action that applies to all strings in $\mathcal{L}(G)$ that belong to $P^{-1}(t')\{\sigma\}$ and to the unobservable continuations of these strings. However $S_P(t)$ may disable unobservable events and thus prevent some of these unobservable continuations. We define

$$L_t = P^{-1}(t')\{\sigma\}\,(S_P(t) \cap E_{uo})^\star \cap \mathcal{L}(G).$$

In words, L_t contains all the strings in $\mathcal{L}(G)$ that are subject to the control action $S_P(t)$. Now remember that a supervisor is admissible if it does not disable uncontrollable events. Hence S_P is admissible if for all $t = t'\sigma \in P[\mathcal{L}(G)]$,

$$E_{uc} \cap \left[\bigcup_{s \in L_t} \Gamma(f(x_0, s))\right] \subseteq S_P(t).$$

Definition A.11 *The language generated by S_P/G is defined recursively as follows*

1. $\epsilon \in \mathcal{L}(S_P/G)$
2. $[(s \in \mathcal{L}(S_P/G))$ *and* $(s\sigma \in \mathcal{L}(G))$ *and* $(\sigma \in S_P[P(s)])] \iff [(s\sigma \in \mathcal{L}(S_P/G))]$.

The language marked by S/G is defined as follows:

$$\mathcal{L}_m(S_P/G) := \mathcal{L}(S_P/G) \cap \mathcal{L}_m(G)\ .$$

A.6 Uncontrollability problem

A.6.1 Dealing with uncontrollable events

In the following is presented the existence result for supervisors in presence of uncontrollable events.

Theorem A.4 (Controllability theorem) *Consider a DES $G = (X, E, f, \Gamma, x_0)$ where $E_{uc} \subseteq E$ is the set of uncontrollable events. Let $K \subseteq \mathcal{L}(G)$, where $K \neq \emptyset$ is the admissible language. Then there exists a supervisor S such that $\mathcal{L}(S/G) = K$ if and only if the controllability condition does hold, i.e.:*

$$\overline{K}E_{uc} \cap \mathcal{L}(G) \subseteq \overline{K} \ .$$

Proof. See [17]. ◁

Remark A.1 *The controllability condition in controllability theorem is intuitive and can be paraphrased as: "if you cannot prevent it, then it should be legal".*

Definition A.12 (controllability) *Let K and $M = \overline{M}$ be languages over set E. Let E_{uc} be a subset of E. K is said to be controllable with respect to M and E_{uc} if and only if*

$$\overline{K}E_{uc} \cap M \subseteq \overline{K} \ .$$

Theorem A.5 (Nonblocking Controllability theorem) *Consider the DES $G = (X, E, f, \Gamma, x_0)$ where $E_{uc} \subseteq E$ is the set of uncontrollable events. Consider the language $K \subseteq \mathcal{L}_m(G)$, where $K \neq \emptyset$ is the admissible language. Then there exists a nonblocking supervisor S for G such that $\mathcal{L}_m(S/G) = K$ and $\mathcal{L}(S/G) = \overline{K}$ if and only if:*

1. *K is controllable with respect to $\mathcal{L}(G)$ and E_{uc}, i.e.:*

$$\overline{K}E_{uc} \cap \mathcal{L}(G) \subseteq \overline{K} \ ,$$

2. *K is $\mathcal{L}_m(G)$-closed, i.e.*

$$K = \overline{K} \cap \mathcal{L}_m(G) \ .$$

Proof. See [17]. ◁

A.6.2 Realization of supervisors

Let us assume that language $K \subseteq \mathcal{L}(G)$ is controllable, then from controllability theorem we know that supervisor S defined by

$$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \left\{\sigma \in E_c \ : \ s\sigma \in \overline{K}\right\}$$

results in

$$\mathcal{L}(S/G) = \overline{K} \ .$$

We need now to build a convenient representation of the function S. Consider now an automaton R that marks the language $\overline{K}$:

$$R = (Y, E, g, \Gamma_R, y_0, Y)$$

where R is trim and

$$\mathcal{L}_m(R) = \mathcal{L}(R) = \overline{K}.$$

If we connect R to G by the product operation, the result $R \times G$ is exactly the behavior we desire for S/G:

$$\begin{aligned}
\mathcal{L}(R \times G) &= \mathcal{L}(R) \cap \mathcal{L}(G) \\
&= \overline{K} \cap \mathcal{L}(G) \\
&= \overline{K} = \mathcal{L}(S/G) \\
\mathcal{L}_m(R \times G) &= \mathcal{L}_m(R) \cap \mathcal{L}_m(G) \\
&= \overline{K} \cap \mathcal{L}_m(G) \\
&= \mathcal{L}(S/G) \cap \mathcal{L}_m(G) = \mathcal{L}(S/G) \ .
\end{aligned}$$

Note that R is defined to have the same event set as G, then $R \parallel G = R \times G$. We will call R the *standard realization* of S.

A.7 Unobservability problem

Consider now the feedback loop in the case of partial event observation. In other words we have to deal with the presence of unobservable events in addition to the presence of uncontrollable events. Clearly unobservable events impose further limitations on the controlled behaviors that can be achieved with P-supervisors. As we did for controllability, we need to introduce the concept of *observability*. Intuitively observability means *"if you cannot differentiate between two strings, then these strings should require the same control action"*, or equivalently *"if you must disable an event after observing a string, then by doing so you should not disable any string that appears in the desired behavior"*. This idea can be formalized as follows:

Definition A.13 (observability) *Let K and $M = \overline{M}$ be languages over set E. Let E_c be a subset of E. Let E_o be another subset of E with P as the corresponding natural projection from $E^\star$ to $E_o^\star$. K is said to be observable with respect to M, P and E_c if for all $s \in \overline{K}$ and for all $\sigma \in E_c$,*

$$(s\sigma \notin \overline{K}) \text{ and } (s\sigma \in M) \Rightarrow P^{-1}\left[P(s)\right]\sigma \cap \overline{K} = \emptyset \ .$$

Theorem A.6 (Controllability and observability theorem) *Consider DES $G = (X, E, f, \Gamma, x_0)$ where $E_{uc} \subseteq E$ is the set of uncontrollable events and $E_o \subseteq E$ is the set of observable events. Let P be the natural projection from $E^\star$ to $E_o^\star$. Consider the language $K \subseteq \mathcal{L}_m(G)$, where $K \neq \emptyset$ is the admissible language. Then there exists a nonblocking P-supervisor S_P for G such that $\mathcal{L}_m(S_P/G) = K$ and $\mathcal{L}(S_P/G) = \overline{K}$ if and only if:*

1. *K is controllable with respect to $\mathcal{L}(G)$ and E_{uc}.*
2. *K is observable with respect to $\mathcal{L}(G)$, P and E_{uc}.*
3. *K is $\mathcal{L}_m(G)$-closed.*

Proof. See [17]. ◁

The reader interested in further results regarding supervision of uncontrollable and unobservable languages is referred to [17] and references therein.

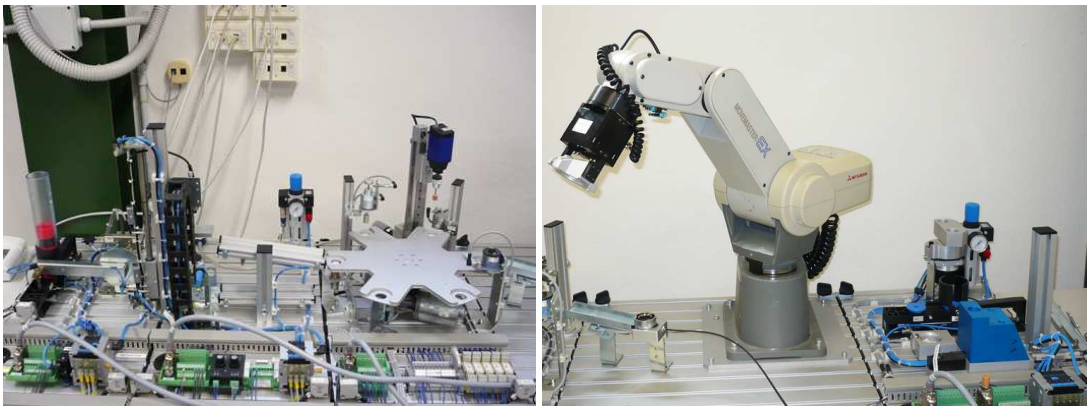
Appendix B

The demonstrator

This appendix describes the FESTO Flexible Manufacturing System (FMS), a didactic setup used to test the architecture proposed in this thesis.

B.1 Testbed description

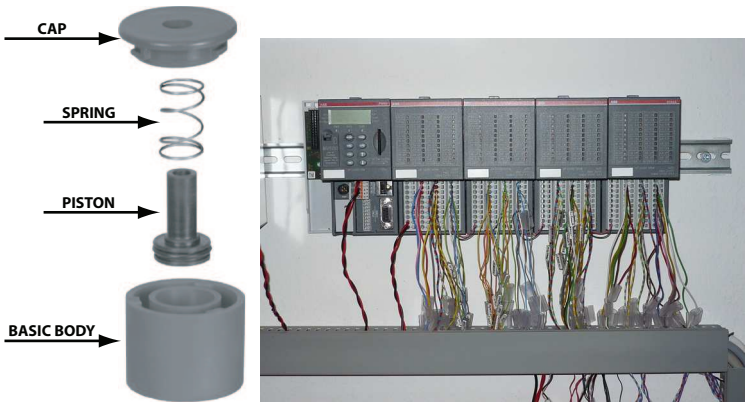
The control architecture based on the *Generalized Actuator* and *Generalize Device* was validated on the testbed of the Laboratory of Automation of University of Bologna (see fig. B.1).



(a) Distribution, testing and processing stations. (b) Assembly station.

Figure B.1: Micro flexible manufacturing system.

The testbed is a miniaturized flexible manufacturing system (FMS) produced by FESTO-DIDACTIC (see fig. B.3); the plant is devote to produce short-stroke cylinders each of them composed by a basic body, a piston, a spring and a cap. In particular the system starts from raw pieces which are worked to realize the bodies and assemblies them with the other parts



(a) Short-stroke cylinders.

(b) PLC.

Figure B.2: Control hardware and short-stroke cylinders.

to obtain the desired cylinder (see fig. B.2(a)). Thanks to the use of different basic bodies it is possible to realize different diameter cylinders. In the following cylinders' bodies will be referred as workpieces.

The FMS is composed by four stations (see fig. B.3): the first station is the **distribution station**, where the workpiece is picked from the raw materials warehouse and moved to the second station, the testing station. In **testing station** the workpiece is measured and its color and height is identified. According to this measurements the workpiece is discarded or moved to the **processing station**; in this station the workpiece is tested to verify if it can be worked or not. If the workpiece positively passes the test, it is drilled and then moved to the last station, the **assembly station**, where workpieces are assembled by a robotic manipulator to realize the cylinder. The control of the FMS is implemented on a ABB PLC, AC500 family equipped with CPU PM581-ETH with four input/output modules DC523 (see fig. B.2(b)).
Exhaustive informations on the FESTO FMS can be find in [33], [35], [34] and [32]. The control software has been developed using the software suite CoDeSys (Controller Development System) [2], a CACSD tool that allows a completely IEC 61131-3 compliance. For more information on the Laboratory of Automation of University of Bologna see [18].

B.1.1 Distribution station

The distribution station (see fig. B.4) fetches the bases of cylinders from an apposite warehouse and moves them to the testing station with the aid of a rotary arm. This station is composed by the following devices:

- **Raw pieces warehouse:** is a FIFO warehouse with the maximum capability of eight pieces: a fiber optic presence sensor furnish an high logic level signal when there are no pieces in the warehouse.

- **Extraction cylinder:** is a single acting pneumatic cylinder whose function is to extract a load from the warehouse. The device is equipped with two inductive sensors sensible to the magnetic field of the piston head. These two limit switch sensors indicates the end positions of the cylinder.

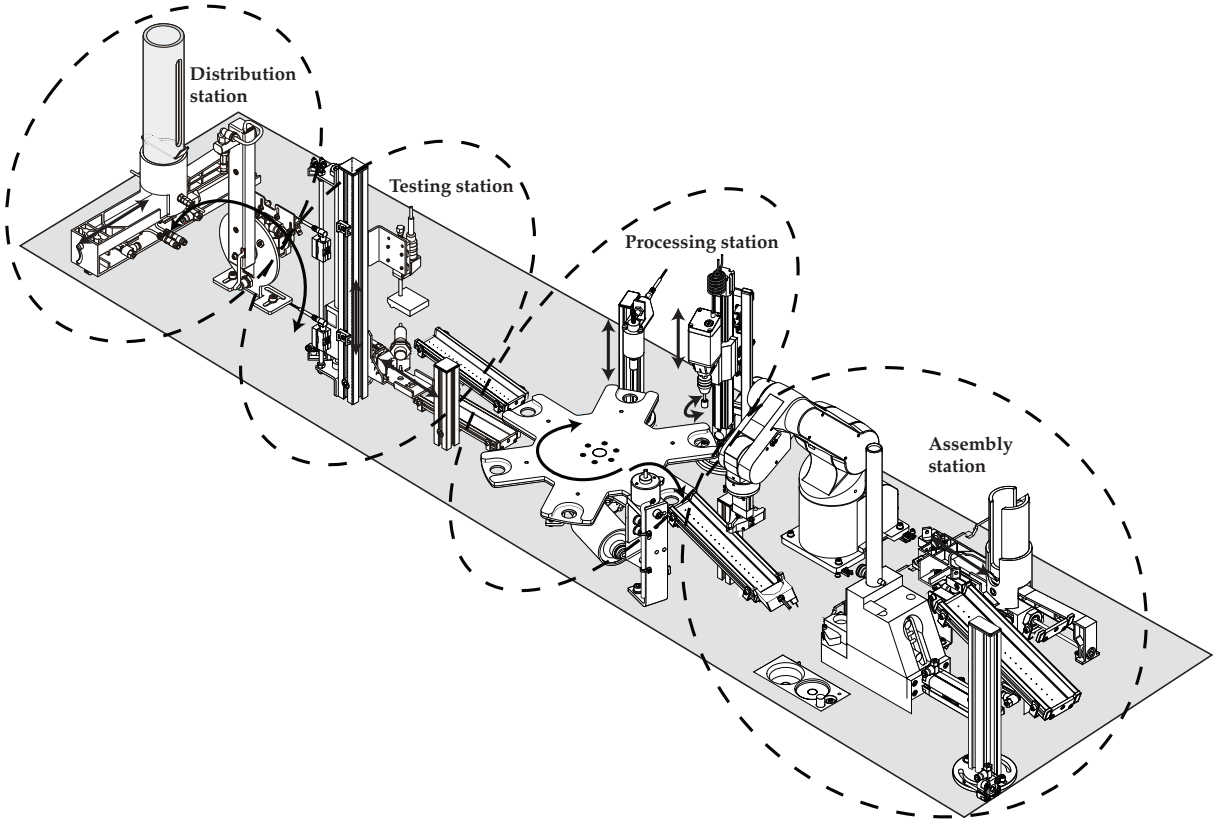


Figure B.3: Micro flexible manufacturing system.

- **Rotary arm:** is a pneumatic rotary manipulator whose movement amplitude can be controlled by mechanical stops. The rotary arm transfers the raw pieces extracted from warehouse to the testing station. The arm is a double acting pneumatic device and its extreme positions are signaled by two electric microswitch activated directly by the arm. To an extremity of the rotary arm there is a suction cup necessary to the capture of the raw pieces by the device. A vacuum sensor indicates when the load is correctly captured.

The usually sequence of operations are:

1. The rotary drive swivels to the position "downstream station" if workpieces are identified in the magazine and the START button is pressed.
2. The ejecting cylinder retracts and pushes a workpieces out of the magazine.
3. The rotary drive swivels to the position "magazine".
4. The vacuum is switched on. When the workpiece is securely held, a vacuum switch switches.
5. The ejecting cylinder advances and releases the workpiece.
6. The rotary drive swivels to the position "downstream station".
7. The vacuum is switched off.

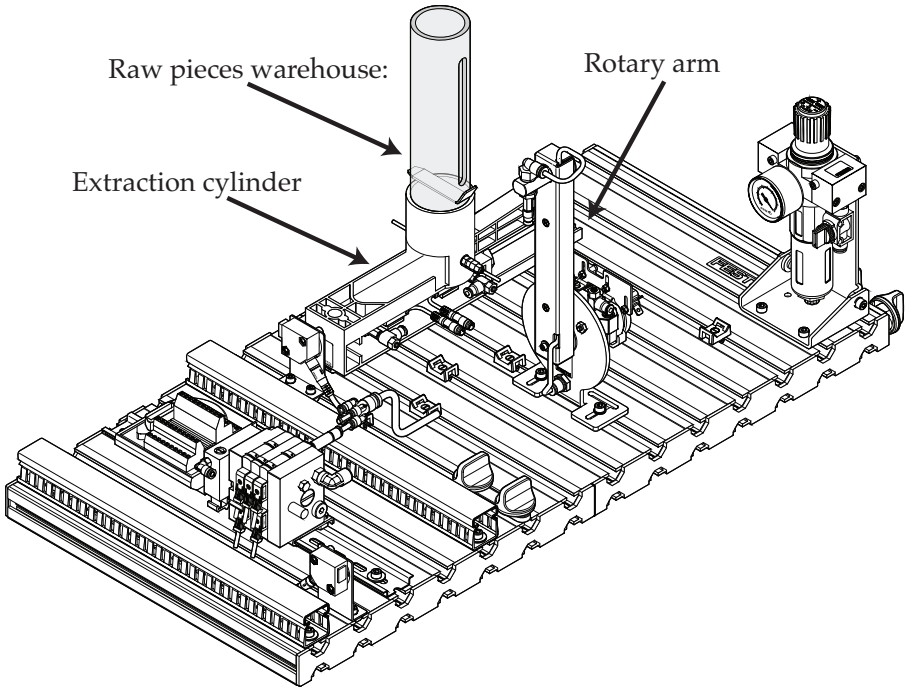


Figure B.4: Distribution Station layout

Signals	Meaning when active	Type
EMPTYWAREHOUSE	warehouse is empty	IN
CYLINDEREXTRACTSLOADFROMWAREHOUSE	extract workpiece from warehouse	OUT
CYLINDEREXTRACTIONLOADINEXTENSIVEPOSITION	extraction link extended position	IN
CYLINDEREXTRACTIONLOADINRETROACTIVEPOSITION	extraction link retracted position	IN
LIGHTEMPTYWAREHOUSE	empty warehouse alarm	OUT
ROTARYMAKERVSWAREHOUSE	Rotary link moving to warehouse	OUT
ROTARYMAKERVSVERIFICATION	Rotary link moving to verification station	OUT
ROTARYMAKERINPOSITIONWAREHOUSE	Rotary link reaches warehouse	IN
ROTARYMAKERINPOSITIONVERIFICATION	Rotary link reaches verification station	IN
VACUUMGENERATOR	Vacuum generation	OUT
VACUUMGENERATOROK	Vacuum generator grasp workpiece	IN
EXPULSIONAIRVACUUM	Expelled workpiece from vacuum generator	OUT

Table B.1: List of signals used in distribution station.

8. The rotary drive swivels to the position “magazine”.

B.1.2 Testing station

The testing station (see fig. B.5) is devoted to checks the colour and the height of a base and, according to the user requests (by the control panel), it decides to send or not the base to the processing station. This station is composed by the following devices.

- **Testing module:** its purpose is to recognize the kind of raw pieces. It includes a capacitive sensor that furnish an high level logic signal if a load is present and a colour sensor to

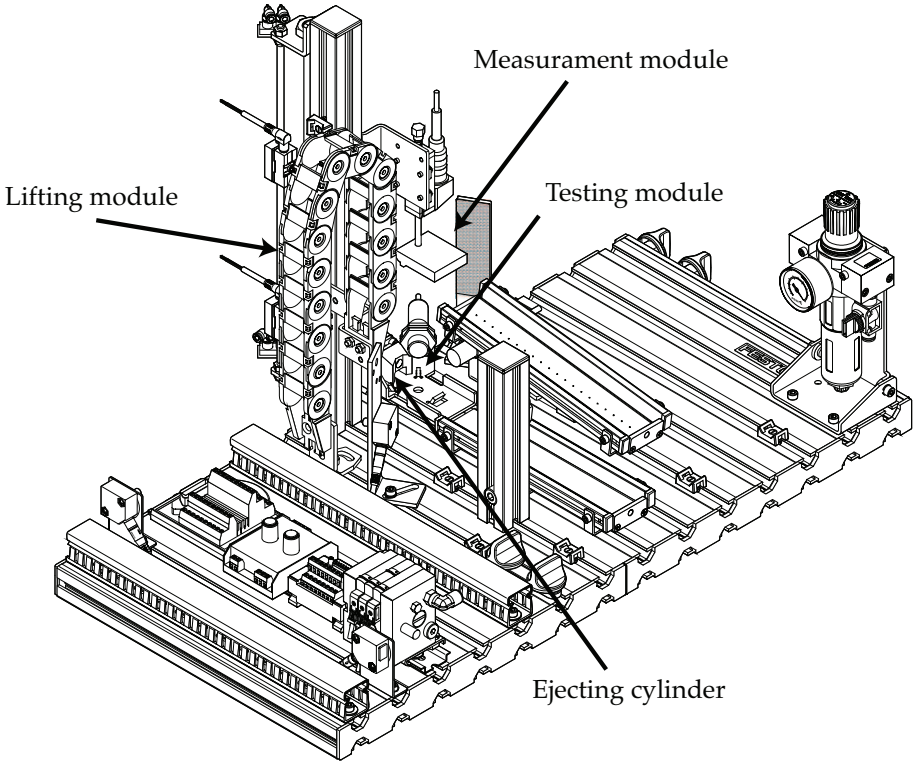


Figure B.5: Testing station layout

sense the colour of the piece. This is a diffuse light sensor which generate an infrared radiation and change it's state when this radiation returns to the sensor meaning that the load present in the testing station is not black.

- **Lifting module:** depending by the raw pieces colour and by the user's specification, the module must lift the raw pieces to the measurement module or stay in the low position to allow the piece expulsion. It is a rodless linear pneumatic drive utilizing magnetic force transmission without mechanical connection. Two inductive sensor are used to indicate the ends position of the lifting module. A retro-reflective sensor is used to sense the presence of the rotary arm into the trajectory of the lifting module in order to avoid a mechanical interference. A single acting pneumatic cylinder mounted on the elevator platform perform the expulsion of the load from the testing station, an inductive sensor indicates its retract end-position. The expulsion of the load toward the processing station is possible thanks to an air guide that reduce the friction of the load along the exit ramp.

- **Measurement module:** the purpose of this module is to recognize the height of the raw pieces presents in the station. This measurement must respects that of colour otherwise the piece cannot proceed to the processing station and must be removed from the productive line. This module consists of an analogue displacement sensor that is a conductive plastic potentiometer which furnish, as output, an analogue voltage which is sent to a analogue comparator. The comparator has three digital output which are at high logic level depending by the value of two threshold voltage set by the user. In this way it's possible to have a digital signal that inform on the presence of a Red/Silver (which have the same height) piece or Black piece.

Signal	Meaning when active	Type
READYLOADFORVERIFICATION	Workpiece in verification station	IN
COLOURMEASUREMENT	Red or silver workpiece	IN
TOLIFTCYLINDERTOMEASURELOAD	Move the lift upward	OUT
TOLOWERCYLINDERTOMEASURELOAD	Move the lift downward	OUT
CYLINDERUPTOMEASURELOAD	The lift is in upward position	IN
CYLINDERDOWNTOMEASURELOAD	The lift is in upward position	IN
TOEXTENDCYLINDEROFEXTRACTIONVSGUIDE	Extraction cylinder expel a workpiece	OUT
CYLINDEROFEXTRACTIONINRETROACTIVEPOSITION	Extraction cylinder in retractive position	IN
AIRCUSHION	Activate air cushion	OUT
MEASUREMENTNOTOK	Bad workpiece	IN

Table B.2: List of signals used in testing station.

The usually sequence of operations are:

1. Determine the colour and material of the workpiece.
2. Lifting cylinder to be raised.
3. Measurement of the workpiece height

If testing result is OK the sequence of operation are:

1. Switch on the air cushioned slide.
2. Ejecting cylinder to advance.
3. Ejecting cylinder to retract.
4. Switch off the air cushioned slide.
5. Lifting cylinder to be lowered.
6. Initial position.

If testing result is not OK the usually sequence of operation are:

1. Lifting cylinder to be lowered.
2. Ejecting cylinder to be advanced.
3. Ejecting cylinder to retract.
4. Initial position.

B.1.3 Processing Station

The processing station (see fig. B.6) function is to perform a processing simulation on the raw pieces: a rotary table is used to move the raw pieces into three different working module. First the piece is tested to verify if it can be worked or not: a raw piece can be worked only if it is correct oriented. If the base successfully passes the test, it is drilled to create the hole for the spring and the piston. After this operation the load is moved toward the expelling module for being transfered at the assembly station. This station is only electrically actuated; no pneumatic devices are present.

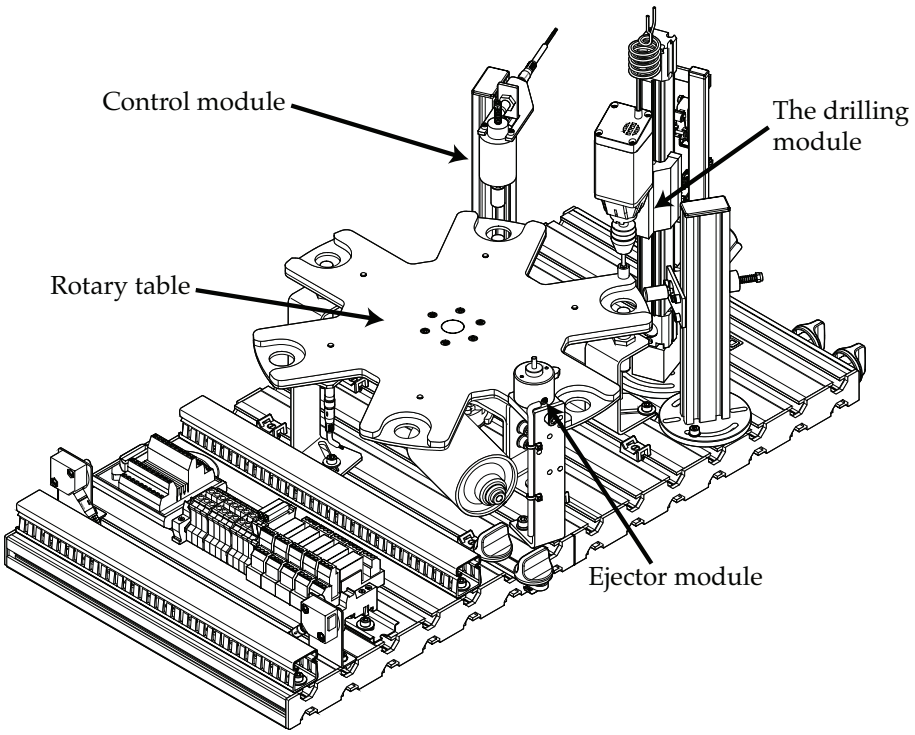


Figure B.6: Processing station layout

- **Rotary table:** the rotary table allow the movement of the raw pieces toward the different working module. The table is actuated by a 24 Volts DC-motor and has six different places for the containment of the pieces. The six positions are then 60 degrees spaced; below the table there is an inductive sensor that changes its state when the table is aligned. Three capacitive proximity sensors are used to sense the presence of a load in the first three position of the table (initial position, control module and drilling module).
- **Control module:** this module check the orientation of the pieces (remember that the pieces are just worked because the system does not perform any real operation). It consists of an electrically actuated single acting cylinder; when the actuation command is high, an electromagnet force the cylinder to move down while when the actuation command is low, a return spring force the cylinder to stay in up position. An inductive sensor give an high logic level signal when the cylinder is completely descended. This means that the load is correct oriented (with the hole facing up).
- **The drilling module:** the drilling module is used to simulate the polishing of the hole of the workpiece. An electrical clamping device retains the workpiece. The feed and return actions of the drilling machine are effected by means of a linear axis with toothed belt drive. An electrical gear motor drives the linear axis and a relay circuit is used to activate the motor. The motor of the drilling machine is operated via 24 Volts DC and the speed is not adjustable. The end position sensing is effected by means of electrical limit switches. Approaching of the limit switches causes a reversal of the direction of movement of the linear axis. The drilling machine is equipped with a drilling tool moved by a 24 Volts DC-motor.
- **Ejector module:** after the polishing of the pieces, the table must turn so that the load can

Signal	Meaning when active	Type
AlignementRotaryTableWithPositioning	Rotary table is aligned	IN
RotaryTableMotor	Move the rotary table	OUT
InControlLoadInWrongPositionToBeDrilled	The workpiece is upside-down	IN
AvailableLoadInControlPositioning	The workpiece is in the testing module	IN
ToLowerCylinderToInspectLoad	Test the workpiece in testing unit	OUT
AvailableLoadInDrillingPositioning	The workpiece is in the drilling module	IN
BlockingCylinderForwardInDrillingPositioning	Lock the workpiece in the drilling module	OUT
DrillingUnitDown	Drilling machine has reached the downward limit	IN
DrillingUnitUp	Drilling machine has reached the upward limit	IN
ToLowerDrillingUnit	Move downward the drilling machine	OUT
ToLiftDrillingUnit	Move upward the drilling machine	OUT
DrillingUnitClockWise	Select clockwise rotation	OUT
DrillingUnitUnClockWise	Select counter-clockwise rotation	OUT
DrillingUnitActive	Activate rotation of drilling machine	OUT
ExpellingLeverActive	Expel the workpiece in pushing-out module	OUT
LightUpsideDownLoadInExpelling	Activate alarm of upside-down workpiece	OUT

Table B.3: List of signals used in processing station.

be sent into the assembly station by the ejector module. The module is an electrically actuated lever; when the actuation command is high the lever expels the load while removing the command the lever returns on its initial position. As mentione before, in this position there is no a presence sensor for the load.

The usually sequence of operations are:

1. The rotary indexing table is rotated by $60°$, if a workpiece is detected in the workpiece retainer 1 and the START pushbutton is pressed.

2. The solenoid plunger moves downwards and checks whether the workpiece is inserted with the opening facing upwards. The rotary indexing table is rotated by $60¡$ if the result of the check is OK.

3. The clamping device clamps the workpiece. The motor of the drilling machine is switched on. The linear axis moves the drilling machine downwards.

4. When the drilling machine has reached its lower position, it is moved to its upper stop again by the linear axis.

5. The motor of the drilling machine is switched off and the clamping device is retracted. The rotary indexing table is rotated by $60°$.

6. The electrical sorting gate passes on the workpiece to a subsequent station.

B.1.4 Assembly station

This station (see fig. B.7) supplies the components of the cylinder for the assembly process. The assembly is performed by a five degrees of freedom manipulator.

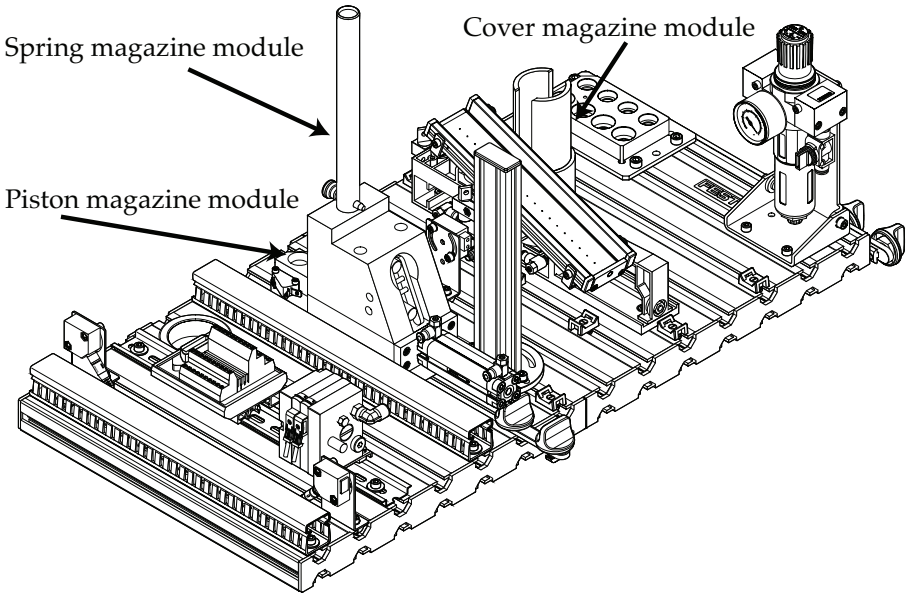


Figure B.7: Assembly station layout

- **Robot:** the robot perform all the assembly operation to obtain the desired final object. Its function is to take the different pieces by the different magazines and to mount this pieces in the correct order into the basic body. The robot is a Mitsubishi RV-M1 five degrees of freedom manipulator which have a proper control module for the planning of its movements and to power its motors. Thanks to the use of different basic bodies, it is possible to produce short-stroke cylinders of different piston diameters. During the assembly operation the piece is blocked thanks to a pneumatic single acting piston which is retracted if the actuation command is high.

- **Spring magazine module:** a single acting cylinder pushes the springs out of a slim magazine which can contain up to 8 springs. The cylinder is normally extracted, when the actuation command is high the cylinder retracts so a spring can go into the dedicated place. When the actuation command is then removed, the spring is available for being captured by the manipulator. The extreme end-positions of the cylinder are signalized by two inductive sensors.

- **Piston magazine module:** two slim magazines (with capacity of up to 4 pistons) contains the two different type of pistons. The extraction of the correct piston is performed by a double acting cylinder which moves a semicircular platform; at the both ends of the structure there is a circular hole, when this hole is aligned with the respective magazine, a

piston from the warehouse falls into the hole. This piston can be expelled by a subsequent rotation of the platform. The extreme end-positions of the platform are indicated by two inductive sensors.

- **Cover magazine module:** a single acting cylinder pushes the springs out of the cover magazine which can contain up to 8 covers. The device is equipped with two inductive sensors which indicates the end positions of the cylinder. A fiber optic presence sensor furnish an high logic signal when there are no covers in the warehouse.

The usually sequence of operation are:

1. If a workpiece "body" is detected in the retainer and the START pushbutton at the robot controller is actuated the body is picked up by the robot.
2. The body is transported to the Assembly retainer module and placed at the "change gripper" position.
3. The colour of the body is determined.
4. The body is picked up and the orientation is checked.

If there is a black workpiece the operation are:

1. The body is placed in the correct orientation in the "assembly" position.
2. A metallic piston is picked up at the pallet. The piston is inserted in the body.
3. The robot checks whether a spring is available. If it is, the spring is picked up and placed on the piston.
4. The robot checks whether a cap is available. If it is, the cap is picked up and placed on the bolt of the Assembly retainer module. The orientation of the cap is checked.
5. The cap is placed in the correct orientation on the body. The cap is fixed by means of rotation.
6. The finished pneumatic cylinder is placed on the slide.

If there is a red or silver workpiece the operation are:

1. The body is placed in the correct orientation in the "assembly" position.
2. A black piston is picked up at the pallet. The piston is inserted in the body.
3. The robot checks whether a spring is available or not. If it is, the spring is picked up and placed on the piston.
4. The robot checks whether a cap is available or not. If it is, the cap is picked up and placed on the bolt of the Assembly retainer module. The orientation of the cap is checked.
5. The cap is placed in the correct orientation on the body. The cap is fixed by means of rotation.
6. The finished pneumatic cylinder is placed on the slide.

Signal	Meaning when active	Type
ToExtractSpringInAssemblyStation	Spring extraction	OUT
ToExtractSpringInAssemblyStationInExtensivePosition	Spring extraction link extended	IN
ToExtractSpringInAssemblyStationInRetroactivePosition	Spring extraction link retroactive	IN
PistonSelectorGoOnTheRight	Piston warehouse rotates to the right	OUT
PistonSelectorGoOnTheLeft	Piston warehouse rotates to the left	OUT
PistonSelectorIsOnTheRight	Piston warehouse in right position	IN
PistonSelectorIsOnTheLeft	Piston warehouse in left position	IN
ToExtractCoverInAssemblyStationForward	Cap extraction from warehouse	OUT
ToExtractCoverInAssemblyStationInRetroactivePosition	Link cap extraction in retroactive position	IN
ToExtractCoverInAssemblyStationInExtensivePosition	Link cap extraction in extensive position	IN
EmptyCoverHouseInAssemblyStation	Cap warehouse is empty	IN
BlockingCylinderForwardInAssemblyStation	Unlock workpiece	OUT
RobotInPistonWarehose	Robot in piston warehouse position	IN
RobotInSpringWarehouse	Robot in spring warehouse position	IN
RobotInCoverWarehouse	Robot in cap warehouse position	IN
RobotInAssemblyUnit	Robot in assembly position	IN
RobotTakeCurrentLoadToAssemblyInAssemblyUnit	Workpiece assembly	OUT
RobotGoToPistonHouse	Robot grasp piston	OUT
RobotGoToSpringHouse	Robot grasp spring	OUT
RobotGoToCoverHouse	Robot grasp cap	OUT
RobotGoToInitialHouse	Robot move in initial position	OUT

Table B.4: List of signals used in assembly station.

B.2 Part of code of FESTO

In figure B.8 and figure B.9 are reported a little part of code of FESTO with GA approach. Following is reported the code of the GD:

```
CASE GD_Init OF
     1: IF Activated THEN
              DeviceState:=DeviceActivated;
         END_IF;
         IF Deactivated THEN
              DeviceState:=DeviceDeactivated;
         END_IF;
         IF (NOT Activated AND NOT Deactivated) THEN
              DeviceState:=DeviceStopped;
         END_IF;

    0: CASE DeviceState OF

     DeviceDeactivated:
      DeactivationRequest:=FALSE;
       IF (ActivationRequest) THEN
          Activation:=TRUE;
```

```
        Deactivation:=FALSE;
        DeviceTime:=ActivationTime;
        DeviceState:=DeviceInActivation;
    END_IF;

  DeviceInActivation:
    IF (DeactivationRequest) THEN
        ActivationRequest:=FALSE;
        Activation:=FALSE;
        Deactivation:=TRUE;
        DeviceTime:=DeactivationTime;
        DeviceState:=DeviceInDeactivation;
     ELSIF (NOT ActivationRequest) THEN
        Activation:=FALSE;
        Deactivation:=FALSE;
        DeviceTime:=0;
        DeviceState:=DeviceStopped;
     ELSIF (Activated) THEN
            Activation:=FALSE;
            Deactivation:=FALSE;
            ActivationRequest:=FALSE;
            DeviceTime:=0;
            DeviceState:=DeviceActivated;
     END_IF;

  DeviceActivated:
     ActivationRequest:=FALSE;
     IF (DeactivationRequest) THEN
           Activation:=FALSE;
           Deactivation:=TRUE;
           DeviceTime:=DeactivationTime;
           DeviceState:=DeviceInDeactivation;
     END_IF;

  DeviceInDeactivation:
    IF (ActivationRequest) THEN
        DeactivationRequest:=FALSE;
        Activation:=TRUE;
        Deactivation:=FALSE;
        DeviceTime:=ActivationTime;
        DeviceState:=DeviceInActivation;
     ELSIF (NOT DeactivationRequest) THEN
        Activation:=FALSE;
        Deactivation:=FALSE;
        DeviceTime:=0;
        DeviceState:=DeviceStopped;
     ELSIF (Deactivated) THEN
            Activation:=FALSE;
```

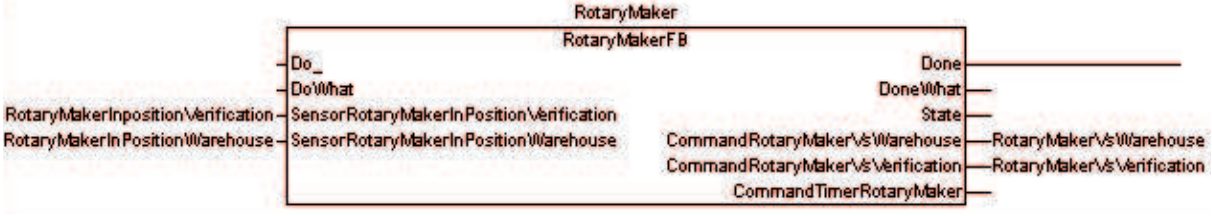
```
              Deactivation:=FALSE;
              DeactivationRequest:=FALSE;
              DeviceTime:=0;
              DeviceState:=DeviceDeactivated;
      END_IF;

    DeviceStopped:
      IF (ActivationRequest AND NOT DeactivationRequest) THEN
          Activation:=TRUE;
          Deactivation:=FALSE;
          DeviceTime:=ActivationTime;
          DeviceState:=DeviceInActivation;
      END_IF;
      IF (DeactivationRequest AND NOT ActivationRequest) THEN
          Activation:=FALSE;
          Deactivation:=TRUE;
          DeviceTime:=DeactivationTime;
          DeviceState:=DeviceInDeactivation;
      END_IF;

   END_CASE;
END_CASE;
(*Gestione Temporizzazione GD*)
IF (ClockT AND (DeviceTime>0)) THEN
       DeviceTime:=DeviceTime-1;
END_IF;
Timeout:=(DeviceTime=0);
(****SEGNALAZIONI DIAGNOSTICA***)
FaultSensorDeviceActivated:=((DeviceState=DeviceDeactivated
 AND Deactivated AND Activated) OR
 (Activation AND NOT Deactivation AND NOT Deactivated AND NOT Activated)
  OR (DeviceState=DeviceActivated AND NOT Deactivated AND NOT Activated))
   AND Timeout;
   FaultSensorDeviceDeactivated:=((DeviceState=DeviceDeactivated AND
  NOT Deactivated AND NOT Activated) OR
  (NOT Activation AND Deactivation AND NOT Deactivated AND NOT Activated)
   OR (DeviceState=DeviceActivated AND  Deactivated AND Activated))
   AND Timeout;
FaultActuator:=((DeviceState=DeviceDeactivated AND (NOT Deactivated
   AND Activated)) OR  (DeviceState=DeviceActivated AND
   (Deactivated AND NOT Activated)) OR
    (Activation AND NOT Deactivation AND Deactivated AND NOT Activated) OR
    (NOT Activation AND Deactivation AND NOT Deactivated AND Activated))
    AND Timeout;

DeviceFault:= FaultSensorDeviceActivated OR
   FaultSensorDeviceDeactivated OR FaultActuator;
```

```
                        DoneWhat:='EndInit';
                    END_IF;

        Init3: IF (NOT DO_) THEN (  *Resetto il DONE quando la politica resetta il DO  *)
                    StateGaExtractionCylinderST:=Ready;
                    State:='Ready';
                    StateInit:=Init1;
                    Done:=FALSE;
                END_IF;
    END_CASE;

Ready: IF (DO_) THEN (  *Aspetto il comando di inizio operazioni
dalla politica  *)
            StateGaExtractionCylinderST:=Busy;
            StateBusy:=Busy1;
            State:='Busy';
        END_IF;

(*In Busy ho 5 sottostati Busy1 (Controllo tipo di comando 'do' e lo
eseguo)  *)
Busy: CASE StateBusy OF
        Busy1:  IF (DoWhat='ExtractionCylinderOut') AND SensorEmptyWarehouse = FALSE THEN (  * estraggo un pezzo dal magazzino  *)
                    CommandCylinderExtractsLoadFromWarehouse:=TRUE;
                    StateBusy:=Busy2;
                END_IF;
                IF (DoWhat='ExtractionCylinderIn') THEN (  * disattivo l'attuatore per far tornare il
                cilindro di estrazione in posizione retratta  *)
                    CommandCylinderExtractsLoadFromWarehouse:=FALSE;
                    StateBusy:=Busy3;
                END_IF;
                IF (DoWhat='ExtractionCylinderOut') AND SensorEmptyWarehouse = TRUE  THEN (  * accendo la
                luce magazzino vuoto  *)
                    CommandLightEmptyWarehouse:=TRUE;
                    Done:=TRUE;
                    DoneWhat:='None';
                    StateBusy:=Busy4;
                END_IF;
                IF (DoWhat='ControlLoadPresence')  THEN (  * accendo la lucemagazzino vuoto  *)
                    CommandLightEmptyWarehouse:=TRUE;
                    StateBusy:=Busy5;
                END_IF;

        Busy2: IF SensorCylinderExtractionLoadInExtensivePosition =TRUE THEN (  *Busy2: aspetto che
        il cilindro di estrazione sia in posizione estesa  *)
                    StateBusy:=Busy4;
                    CommandCylinderExtractsLoadFromWarehouse:=TRUE;
                    Done:=TRUE;
                    DoneWhat:='ExtractionCylinderOut';
                END_IF;

        Busy3: IF SensorCylinderExtractionLoadInRetroactivePosition = TRUE THEN (  * Busy3: aspetto che
        il cilindro di estrazione sia in posizione retratta  *)
                    StateBusy:=Busy4;
                    CommandCylinderExtractsLoadFromWarehouse:=FALSE;
                    DoneWhat:='ExtractionCylinderIn';
                    Done:=TRUE;
                END_IF;

        Busy4: IF (NOT DO_) THEN (  *Busy4: fine operazioni GA  *)
                    Done:=FALSE;
                    StateGaExtractionCylinderST:=Ready;
                    State:='Ready';
                    StateBusy:=Busy1;
                END_IF;

        Busy5: IF PushButtonFullWarehouse = TRUE THEN (  *Busy5: Vedo se  stato riempito il magazzino pezzi  *)
                    CommandLightEmptywarehouse:=FALSE;
                    DoneWhat:='WareHouseFull';
                    StateBusy:=Busy4;
                    Done:=TRUE;
                END_IF;
    END_CASE;
END_CASE;
```

ROTARY MAKER

```
CASE StateGaRotaryMakerST OF

(*Nello stato di Idle apsetto di ricevere il comando per iniziare
```

Figure B.8: Example of GA code on FESTO




```
l'inizializzazione            *)
Idle: IF (Do_ AND DoWhat='InitRotaryMaker1')THEN
            StateGaRotaryMakerST:=Init;
      END_IF;
      IF (Do_ AND DoWhat='InitRotaryMaker2') THEN
            StateGaRotaryMakerST:=Init;
      END_IF;

(*Nello stato di Init eseguo le operazioni di inizializzazione, ho 3
sottostati Init, Init2, Init3 (Fine inizializzazione)                    *)
Init: CASE StateInit OF
        Init1: IF SensorRotaryMakerInPositionWarehouse THEN (              * controllo che il rotary maker sia
        in posizione verifica          *)
                  StateInit:=Init3;
                  Done:=TRUE;
               END_IF;
               IF NOT SensorRotaryMakerInPositionWarehouse) THEN (          * nello stato init  il rotary maker  portato
               in posizione verifica          *)
                  StateInit:=Init2;
                  CommandRotaryMakerVsWarehouse:=TRUE; (       * il rotary maker viene portato in posizione verifica          *)
               END_IF;

        Init2: IF SensorRotaryMakerInPositionWarehouse THEN (              * aspetto che il rotary maker sia arrivato
        in posizione verifica          *)
                  StateInit:=Init3;
                  CommandRotaryMakerVsWarehouse:=FALSE;
                  Done:=TRUE;
               END_IF;

        Init3: IF (NOT DO_) THEN  (      *Resetto il DONE quando la politica resetta il DO          *)
                  StateGaRotaryMakerST:=Idle;
                  State:='Idle';
                  StateInit:=Init4;
                  DoneWhat:='EndInit1';
                  Done:=FALSE;
               END_IF;

        Init4: IF SensorRotaryMakerInPositionVerification THEN (           * controllo che il rotary maker sia
        in posizione verifica          *)
                  StateInit:=Init6;
                  Done:=TRUE;
               END_IF;
               IF (NOT SensorRotaryMakerInPositionVerification) THEN (          * nello stato init  il rotary maker
               portato in posizione verifica          *)
                  StateInit:=Init5;
                  CommandRotaryMakerVsVerification:=TRUE; (        * il rotary maker viene portato in posizione verifica          *)
               END_IF;

        Init5: IF SensorRotaryMakerInPositionVerification THEN (           * aspetto che il rotary maker sia arrivato
        in posizione verifica          *)
                  StateInit:=Init6;
                  CommandRotaryMakerVsVerification:=FALSE;
                  Done:=TRUE;
               END_IF;

        Init6: IF (NOT DO_) THEN  (      *Resetto il DONE quando la politica resetta il DO          *)
                  StateInit:=Init1;
                  StateGaRotaryMakerST:=Ready;
                  State:='Ready';
                  DoneWhat:='EndInit';
                  Done:=FALSE;
               END_IF;
      END_CASE;
```

Figure B.9: Example of GA code on FESTO

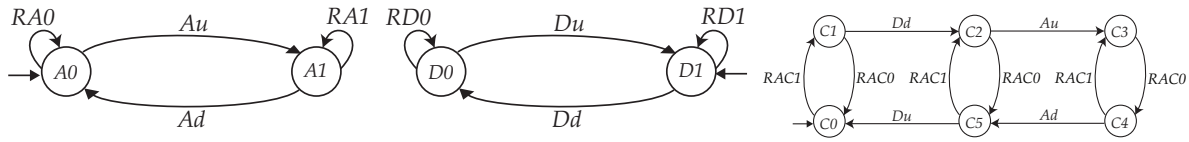
Appendix C

Components models of DES approach

In This appendix the components model deriving from the methodology presented in chapter 5 are reported.

C.1 Examples of model composition

The automaton in figure C.2 is the result of the parallel composition of the automata in figure C.1 (6 states 22 transitions), in this figure are shown the models of sensors and connection constrain. The automata generates from parallel composition have the same number of states of automata which modeling the connection constrain, but for each states there are self loops with value of the sensors. In blue is reported starting from initial state an activation and deactivation sequence, the initial state corresponding to device in deactivate position.



(a) No fault model $G_{L,Anf}$.

(b) No fault model $G_{L,Dnf}$.

(c) Physical Constraint Automaton (PCA) $G_{L,PCA}$.

Figure C.1: Nominal models of the sensors, actuator and Physical Constraint Automaton (PCA) $G_{L,PCA}$.

In figure C.3 the nominal and faulty model of device with faults f_{a1} and f_{d0}.

In figure C.4 the nominal and faulty model of device with faults on sensor A and sensor D.

In figure C.5 the nominal and faulty model of device with faults on actuator, f_1 and f_0.

In figure C.6 and figure C.7 are depicted the change of physical constraints when a fault on sensor D occurred. Figure C.6 is the case of sensor D stuck low and Figure C.7 is the case of sensor D stuck high. When a sensor is stuck low from a physical point of view is like the device do not have the sensor. In this case if sensor D is stuck low, we can remodeling the

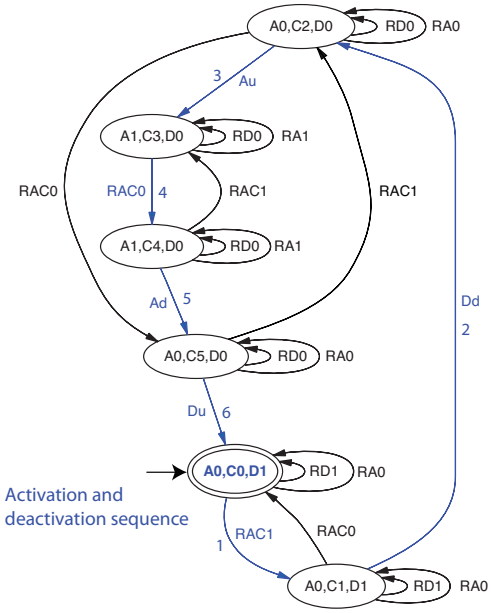
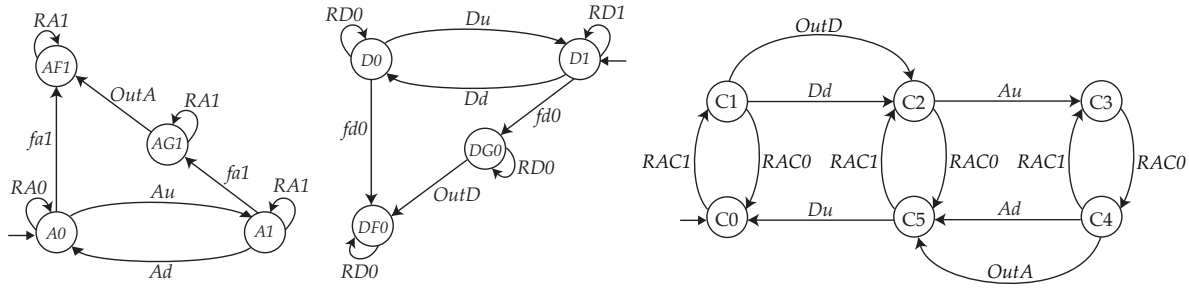


Figure C.2: Composition of nominal sensors and PCA.



(a) Fault model of sensor A, $G_{L,Afa1}$. (b) fault model of sensor D, $G_{L,Dfd0}$. (c) Physical Constraint Automaton with faults, G_{L,PCA_F}.

device with only sensor A, and the device becomes the device of figure C.6. The model of this device is composed by an automata of 4 states, because now we can have only two sensors configuration, 00 configuration and 01 configuration, this model can be obtained, as it shown in figure C.6(b), from model of PCA automaton. If sensor D is stuck low the model has not to generate events Au and Ad, for this reason the model on sensor D when occurs fault f_{d0} evolves and remains in state $DF0$. The PCA automaton do not reach states $C0$ and $C1$ but evolves only in states $C2$, $C3$, $C4$ and $C5$.

In figure C.8(a) and figure C.8(b) are reported how the physical constraints of the device evolves when a fault on actuator occur. It is easy to see that when a fault on actuator occurs the device following the movement of the fault.

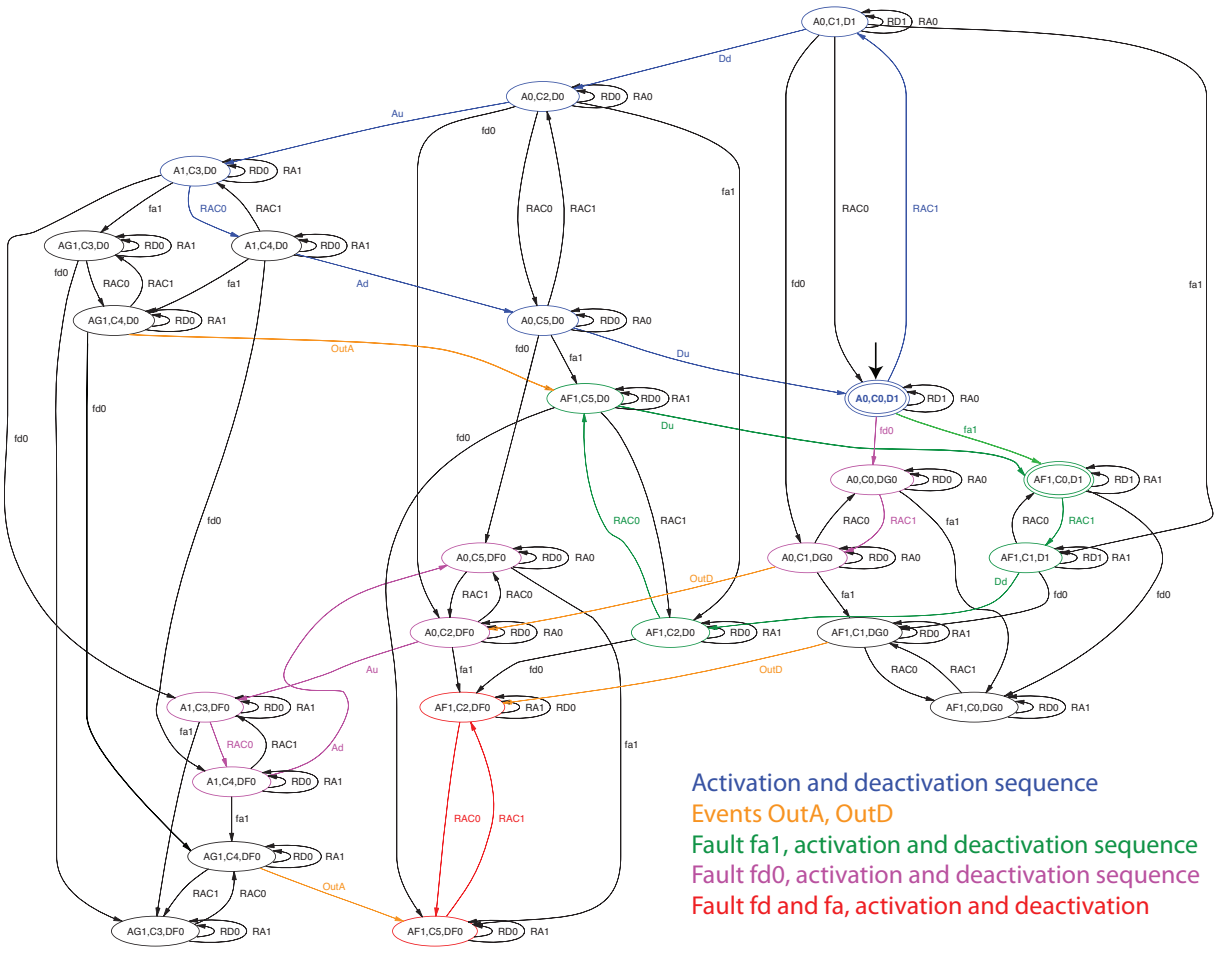
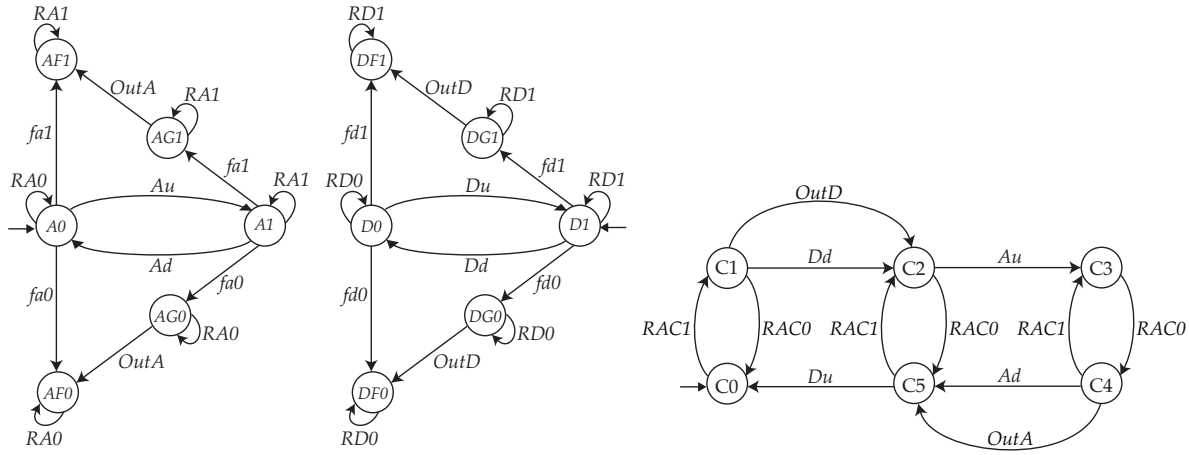


Figure C.3: Composition of nominal and faulty model (faults f_{a1} and f_{d0}).



(a) Fault model of sensor A, $G_{L,Afa}$.

(b) Fault model of sensor D, $G_{L,Dfd}$.

(c) Physical Constraint Automaton with faults, G_{L,PCA_F}.

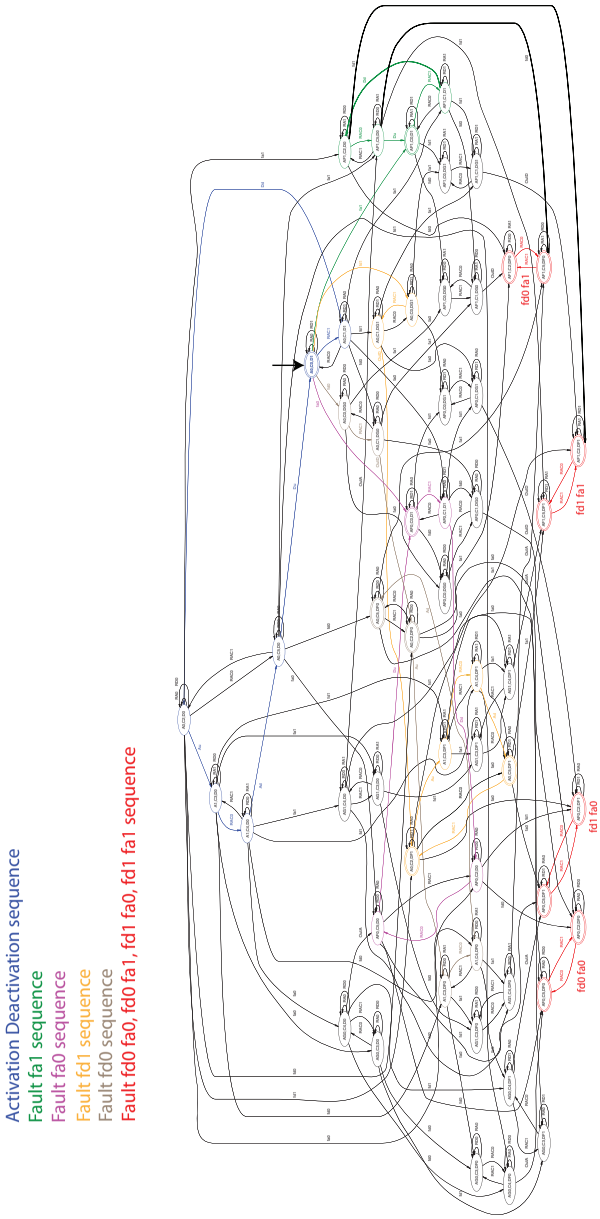
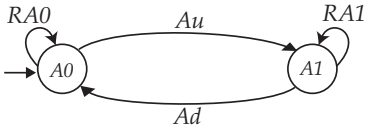
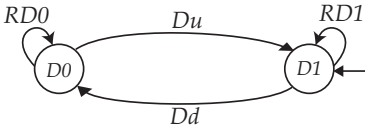


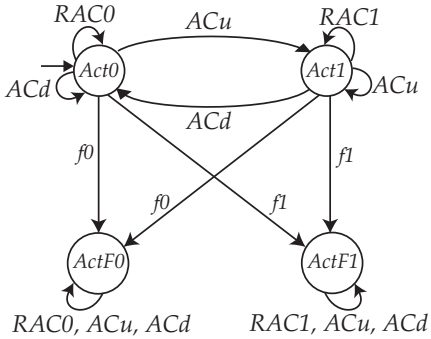
Figure C.4: Composition of nominal and faulty model with faults on sensor D and sensor A.



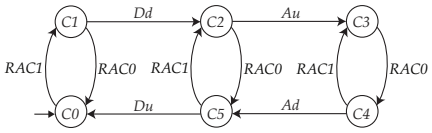
(a) No fault model $G_{L,Anf}$.



(b) No fault model $G_{L,Dnf}$.



(c) Fault model on actuator $G_{L,Act}$.



(d) Physical Constraint Automaton (PCA) $G_{L,PCA}$.

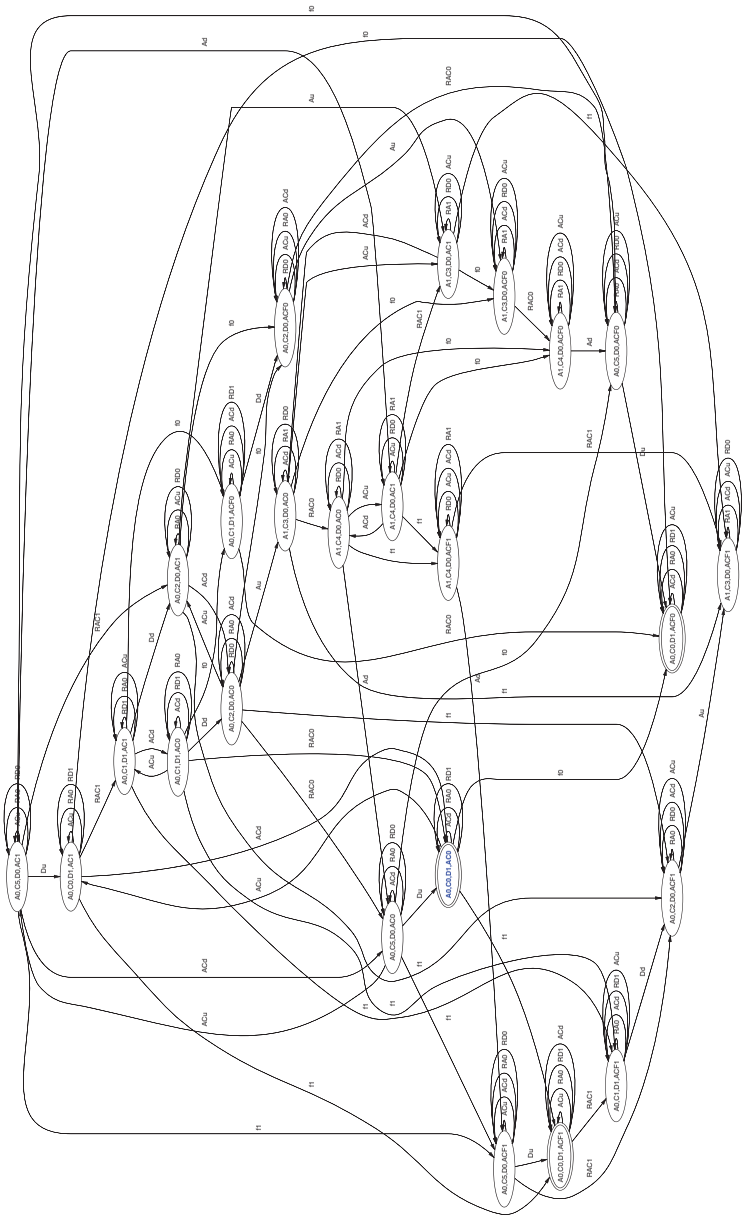
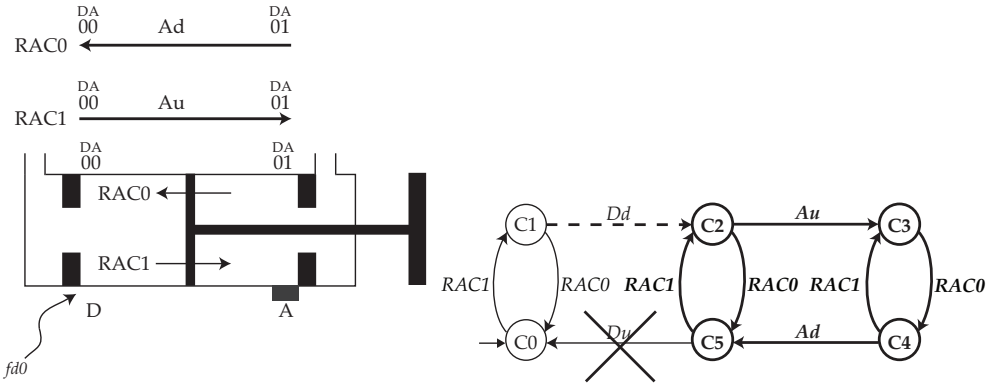
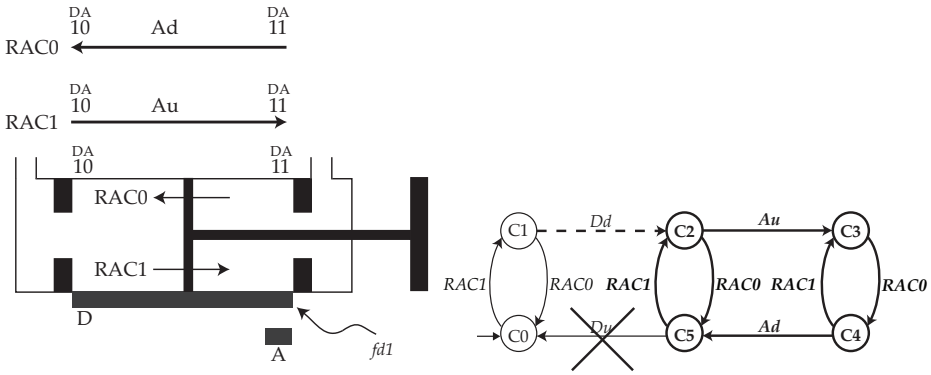


Figure C.5: Composition of nominal and faulty model with faults on actuator.



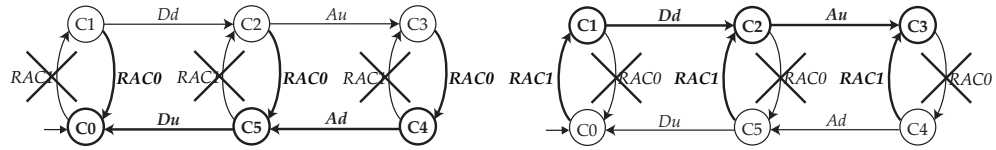
(a) Mechanical configuration fault f_{d0}. (b) New configuration of (PCA) with fault f_{d0}.

Figure C.6: Models of fault f_{d0}.



(a) Mechanical configuration fault f_{d1}. (b) New configuration of (PCA) with fault f_{d1}.

Figure C.7: Models of fault f_{d1}.



(a) New configuration of (PCA) with fault f_0. (b) New configuration of (PCA) with fault f_1.

Figure C.8: Models of actuator faults.

C.2 Control and monitoring of low level devices

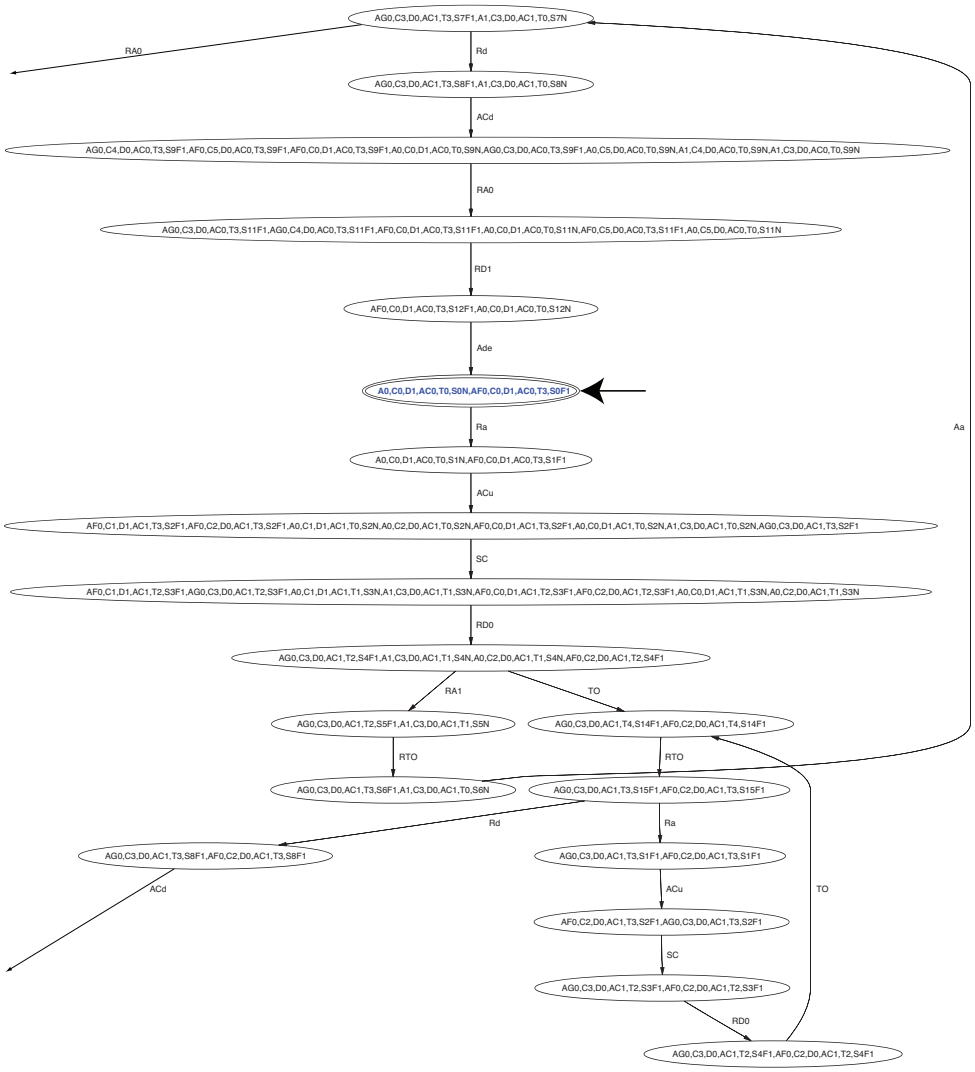


Figure C.9: State concatenation of diagnoser of figure 5.26.

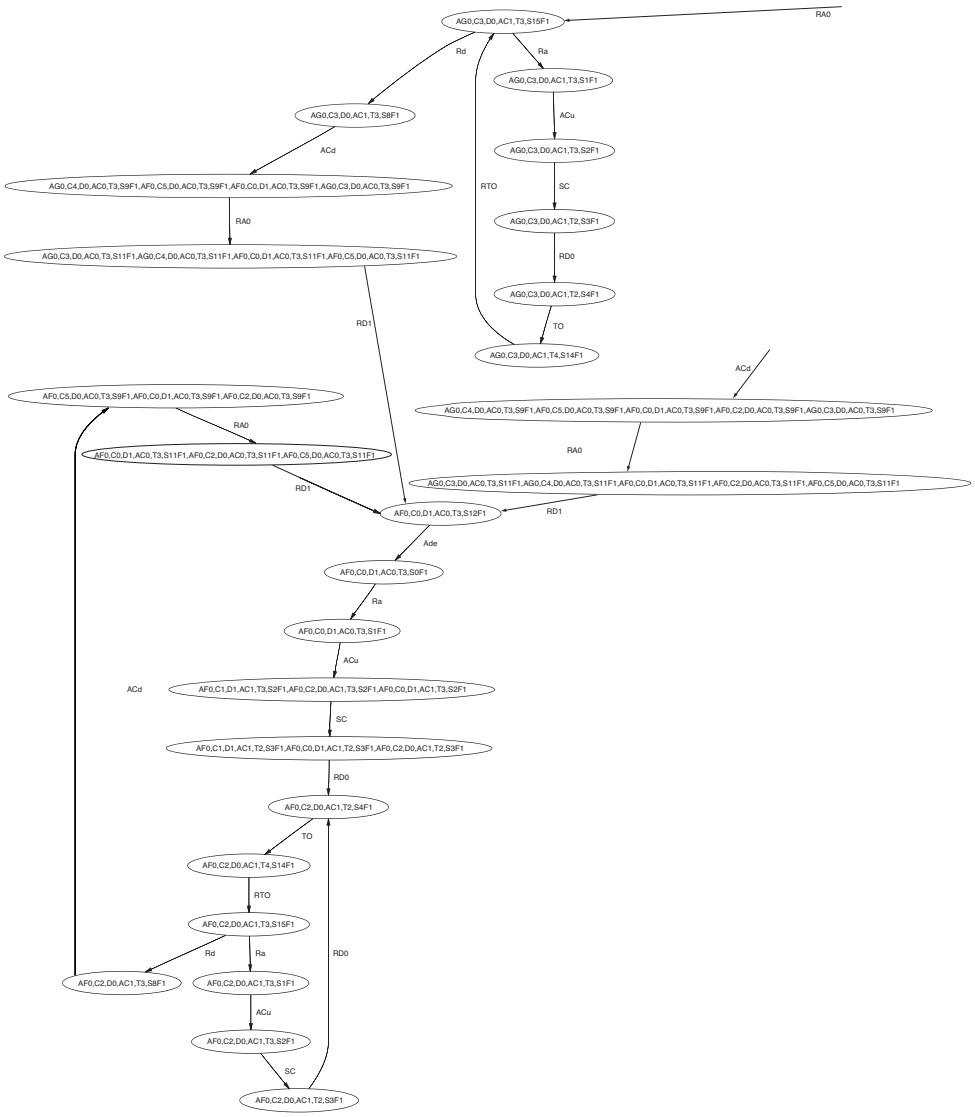


Figure C.10: State concatenation of diagnoser of figure 5.26.

```
Total Diagnoser States = 44

Id = 1
A0,C0,D1,AC0,T0,S0 N
AF0,C0,D1,AC0,T3,S0 F1
Total pairs = 2
Uncertain: F1
Ra -> 2

Id = 2
A0,C0,D1,AC0,T0,S1 N
AF0,C0,D1,AC0,T3,S1 F1
Total pairs = 2
Uncertain: F1
ACu -> 3

Id = 3
AF0,C1,D1,AC1,T3,S2 F1
AF0,C2,D0,AC1,T3,S2 F1
A0,C1,D1,AC1,T0,S2 N
A0,C2,D0,AC1,T0,S2 N
AF0,C0,D1,AC1,T3,S2 F1
A0,C0,D1,AC1,T0,S2 N
A1,C3,D0,AC1,T0,S2 N
AG0,C3,D0,AC1,T3,S2 F1
Total pairs = 8
Uncertain: F1
SC -> 4

Id = 4
AF0,C1,D1,AC1,T2,S3 F1
AG0,C3,D0,AC1,T2,S3 F1
A0,C1,D1,AC1,T1,S3 N
A1,C3,D0,AC1,T1,S3 N
AF0,C0,D1,AC1,T2,S3 F1
AF0,C2,D0,AC1,T2,S3 F1
A0,C0,D1,AC1,T1,S3 N
A0,C2,D0,AC1,T1,S3 N
Total pairs = 8
Uncertain: F1
RD0 -> 5

Id = 5
AG0,C3,D0,AC1,T2,S4 F1
A1,C3,D0,AC1,T1,S4 N
A0,C2,D0,AC1,T1,S4 N
AF0,C2,D0,AC1,T2,S4 F1
Total pairs = 4
Uncertain: F1
TO -> 6
RA1 -> 7

Id = 6
AG0,C3,D0,AC1,T4,S14 F1
AF0,C2,D0,AC1,T4,S14 F1
Total pairs = 2
Certain: F1
RTO -> 8

Id = 7
AG0,C3,D0,AC1,T2,S5 F1
A1,C3,D0,AC1,T1,S5 N
Total pairs = 2
Uncertain: F1
RTO -> 9

Id = 8
AG0,C3,D0,AC1,T3,S15 F1
AF0,C2,D0,AC1,T3,S15 F1
Total pairs = 2
Certain: F1
Ra -> 10
Rd -> 11

Id = 9
AG0,C3,D0,AC1,T3,S6 F1
A1,C3,D0,AC1,T0,S6 N
Total pairs = 2
Uncertain: F1
Aa -> 12

Id = 10
AG0,C3,D0,AC1,T3,S1 F1
AF0,C2,D0,AC1,T3,S1 F1
Total pairs = 2
Certain: F1
ACu -> 13

Id = 11
AG0,C3,D0,AC1,T3,S8 F1
AF0,C2,D0,AC1,T3,S8 F1
Total pairs = 2
Certain: F1
ACd -> 14

Id = 12
AG0,C3,D0,AC1,T3,S7 F1
A1,C3,D0,AC1,T0,S7 N
Total pairs = 2
Uncertain: F1
RA0 -> 15
Rd -> 16

Id = 13
AF0,C2,D0,AC1,T3,S2 F1
AG0,C3,D0,AC1,T3,S2 F1
Total pairs = 2
Certain: F1
SC -> 17

Id = 14
AG0,C4,D0,AC0,T3,S9 F1
AF0,C5,D0,AC0,T3,S9 F1
AF0,C0,D1,AC0,T3,S9 F1
AF0,C2,D0,AC0,T3,S9 F1
AG0,C3,D0,AC0,T3,S9 F1
Total pairs = 5
Certain: F1
RA0 -> 18

Id = 15
AG0,C3,D0,AC1,T3,S15 F1
Total pairs = 1
Certain: F1
Ra -> 19
Rd -> 20

Id = 16
AG0,C3,D0,AC1,T3,S8 F1
A1,C3,D0,AC1,T0,S8 N
Total pairs = 2
Uncertain: F1
ACd -> 21

Id = 17
AG0,C3,D0,AC1,T2,S3 F1
AF0,C2,D0,AC1,T2,S3 F1
Total pairs = 2
Certain: F1
RD0 -> 22

Id = 18
AG0,C3,D0,AC0,T3,S11 F1
AG0,C4,D0,AC0,T3,S11 F1
AF0,C0,D1,AC0,T3,S11 F1
AF0,C2,D0,AC0,T3,S11 F1
AF0,C5,D0,AC0,T3,S11 F1
Total pairs = 5
Certain: F1
RD1 -> 23

Id = 19
AG0,C3,D0,AC1,T3,S1 F1
Total pairs = 1
Certain: F1
ACu -> 24

Id = 20
AG0,C3,D0,AC1,T3,S8 F1
Total pairs = 1
Certain: F1
ACd -> 25

Id = 21
AG0,C4,D0,AC0,T3,S9 F1
AF0,C5,D0,AC0,T3,S9 F1
AF0,C0,D1,AC0,T3,S9 F1
A0,C0,D1,AC0,T0,S9 N
AG0,C3,D0,AC0,T3,S9 F1
A0,C5,D0,AC0,T0,S9 N
A1,C4,D0,AC0,T0,S9 N
A1,C3,D0,AC0,T0,S9 N
Total pairs = 8
Uncertain: F1
RA0 -> 26
```

Figure C.11: List of state concatenation of diagnoser of figure 5.26.

```
Total Diagnoser States = 44

Id = 22
AG0,C3,D0,AC1,T2,S4 F1
AF0,C2,D0,AC1,T2,S4 F1
Total pairs = 2
Certain: F1
TO -> 6

Id = 23
AF0,C0,D1,AC0,T3,S12 F1
Total pairs = 1
Certain: F1
Ade -> 27

Id = 24
AG0,C3,D0,AC1,T3,S2 F1
Total pairs = 1
Certain: F1
SC -> 28

Id = 25
AG0,C4,D0,AC0,T3,S9 F1
AF0,C5,D0,AC0,T3,S9 F1
AF0,C0,D1,AC0,T3,S9 F1
AG0,C3,D0,AC0,T3,S9 F1
Total pairs = 4
Certain: F1
RA0 -> 29

Id = 26
AG0,C3,D0,AC0,T3,S11 F1
AG0,C4,D0,AC0,T3,S11 F1
AF0,C0,D1,AC0,T3,S11 F1
A0,C0,D1,AC0,T0,S11 N
AF0,C5,D0,AC0,T3,S11 F1
A0,C5,D0,AC0,T0,S11 N
Total pairs = 6
Uncertain: F1
RD1 -> 30

Id = 27
AF0,C0,D1,AC0,T3,S0 F1
Total pairs = 1
Certain: F1
Ra -> 31

Id = 28
AG0,C3,D0,AC1,T2,S3 F1
Total pairs = 1
Certain: F1
RD0 -> 32

Id = 29
AG0,C3,D0,AC0,T3,S11 F1
AG0,C4,D0,AC0,T3,S11 F1
AF0,C0,D1,AC0,T3,S11 F1
AF0,C5,D0,AC0,T3,S11 F1
Total pairs = 4
Certain: F1
RD1 -> 23

Id = 30
AF0,C0,D1,AC0,T3,S12 F1
A0,C0,D1,AC0,T0,S12 N
Total pairs = 2
Uncertain: F1
Ade -> 1

Id = 31
AF0,C0,D1,AC0,T3,S1 F1
Total pairs = 1
Certain: F1
ACu -> 33

Id = 32
AG0,C3,D0,AC1,T2,S4 F1
Total pairs = 1
Certain: F1
TO -> 34

Id = 33
AF0,C1,D1,AC1,T3,S2 F1
AF0,C2,D0,AC1,T3,S2 F1
AF0,C0,D1,AC1,T3,S2 F1
Total pairs = 3
Certain: F1
SC -> 35

Id = 34
AG0,C3,D0,AC1,T4,S14 F1
Total pairs = 1
Certain: F1
RTO -> 15

Id = 35
AF0,C1,D1,AC1,T2,S3 F1
AF0,C0,D1,AC1,T2,S3 F1
AF0,C2,D0,AC1,T2,S3 F1
Total pairs = 3
Certain: F1
RD0 -> 36

Id = 36
AF0,C2,D0,AC1,T2,S4 F1
Total pairs = 1
Certain: F1
TO -> 37

Id = 37
AF0,C2,D0,AC1,T4,S14 F1
Total pairs = 1
Certain: F1
RTO -> 38

Id = 38
AF0,C2,D0,AC1,T3,S15 F1
Total pairs = 1
Certain: F1
Ra -> 39
Rd -> 40

Id = 39
AF0,C2,D0,AC1,T3,S1 F1
Total pairs = 1
Certain: F1
ACu -> 41

Id = 40
AF0,C2,D0,AC1,T3,S8 F1
Total pairs = 1
Certain: F1
ACd -> 42

Id = 41
AF0,C2,D0,AC1,T3,S2 F1
Total pairs = 1
Certain: F1
SC -> 43

Id = 42
AF0,C5,D0,AC0,T3,S9 F1
AF0,C0,D1,AC0,T3,S9 F1
AF0,C2,D0,AC0,T3,S9 F1
Total pairs = 3
Certain: F1
RA0 -> 44

Id = 43
AF0,C2,D0,AC1,T2,S3 F1
Total pairs = 1
Certain: F1
RD0 -> 36

Id = 44
AF0,C0,D1,AC0,T3,S11 F1
AF0,C2,D0,AC0,T3,S11 F1
AF0,C5,D0,AC0,T3,S11 F1
Total pairs = 3
Certain: F1
RD1 -> 23
```

Figure C.12: List of state concatenation of diagnoser of figure 5.26.

Name	Means	observable	controllable
Du	Rising signal on sensor D	uo	uc
Dd	Falling signal on sensor D	uo	uc
$RD0$	Signal value on sensor D is low	o	c
$RD1$	Signal value on sensor D is high	o	c
$fd0$	Fault on sensor D, stuck low	uo	uc
$fd1$	Fault on sensor D, stuck high	uo	uc
Au	Rising signal on sensor A	uo	uc
Ad	Falling signal on sensor A	uo	uc
$RA0$	Signal value on sensor A is low	o	c
$RA1$	Signal value on sensor A is high	o	c
$fa0$	Fault on sensor A, stuck low	uo	uc
$fa1$	Fault on sensor A, stuck high	uo	uc
ACu	Rising signal on actuator	o	c
ACd	Falling signal on actuator	o	c
$RAC0$	Movement of actuator to deactivation state	uo	uc
$RAC1$	Movement of actuator to activation state	uo	uc
$f0$	Fault on actuator, blocked low	uo	uc
$f1$	Fault on actuator, blocked high	uo	uc
Ra	Request of activation	o	c
Rde	Request of deactivation	o	c
Aa	Answer of activation	o	c
Ade	Answer of deactivation	o	c
SC	Command to start count of the timer	o	c
RTO	Command to reset the timer	o	c
TO	Timeout of the timer	o	c
$G_{L,Anf}$	Nominal model of sensor A		
$G_{L,Dnf}$	Nominal model of sensor D		
$G_{L,Actnf}$	Nominal model of actuator		
$G_{L,PCA}$	Model of the Physical Constraint Automaton (PCA)		
$G_{L,Afa0}$	Model of sensor A with fault $fa0$		
G_{L,PCA_A}	Model of the Physical Constraint Automaton (PCA) with fault $fa0$		
$G_{L,Tfa0}$	Model of timer		
$G_{L,ConNom}$	Specification of nominal control for the device		
$G_{L,ConDiag}$	Specification of the control with diagnosis for the device		
$G_{L,CompNom}$	Composition of nominal sensors, actuator and PCA		
$G_{L,Compfa0}$	Composition of nominal and $fa0$ faulty model for the device		
$G_{L,DevNom}$	Composition controlled nominal single acting device		
$G_{L,Devfa0}$	Composition controlled faulty single acting device		
$G_{L,Totfa0}$	Composition controlled and diagnosis faulty single acting device		

Table C.1: List of events and automata models.

Bibliography

[1] Sysml. http://www.sysml.org.

[2] 3s Software. 3s-software.

[3] C. Alexander. *A pattern language: Towns, Buildings, Construction*. Oxford university Press, 1997.

[4] B. Vogel-Heuser, D. Friedrich, U. Katzke, D. Witsch. Usability and benefits of uml for plant automation - some research results. *atp International No. 1*, 3 (2005).

[5] K. Beck and W. Cunningham. *OOPSLA-87 Technical Report No. CR-87-43 Using Pattern Languages for Object-Oriented Programs*. 1987.

[6] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki. *Diagnosis and fault-tolerant control*. Springer-Verlag, 2003.

[7] W. Bolton. *Programmable Logic Controllers*. Elsevier, 2006.

[8] M. Bonfè and C. Fantuzzi. Mechatronic objects encapsulation in iec 1131-3 norm. *IEEE International conference on Control Applications*, pages 598 – 603, 2000.

[9] M. Bonfè and C. Fantuzzi. Object-oriented approach to plc software design for a manufacture machinery using iec 61131-3 norm languages. *IEEE International conference on Control Application*, 2:850 – 852, 2001.

[10] M. Bonfè and C. Fantuzzi. Design and verification of mechatronic object-oriented models for industrial control systems. *IEEE International conference on Emerging Technologies and Factory Automation*, 2:253 – 260, 2003.

[11] M. Bonfè and C. Fantuzzi. Application of object-oriented modeling tools to design the logic control system of a packaging machine. *IEEE International conference on Control application Industrial Informatics*, pages 506 – 574, 2004.

[12] M. Bonfè and C. Fantuzzi. A practical approach to object-oriented modeling of logic control system for industrial applications. *IEEE International conference on Decision and Control*, 1:980 – 985, 2004.

[13] M. Bonfè, C. Fantuzzi, and C. Secchi. Behavioural inheritance in object-oriented models for mechatronic systems. *International Journal of Manufacturing Research*, 1(4):421 – 441, 2006.

[14] C. Bonivento, A. Paoli, and M. Sartini. Parameters optimization in a production line using genetic algorithms. *International Conference on Computational Intelligence for Modelling, Control and Automation,*, 2008.

[15] D. Brandl. *Design patterns for flexible manufacturing*. ISA, 2006.

[16] E. Carpanzano, A. Cataldo, and M. Doná. Rapid prototyping test-bed of logic control solution for reconfigurable manufacturing systems. *IEEE International conference on Emerging Technologies and Factory Automation*, 2:621 – 628, 2005.

[17] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems - Second Edition*. Springer, 2007.

[18] CASY-DEIS. Auomation engineering laboratory.

[19] G. Cloutier and J. J. Paques. Gemma, the complementary tool of the grafcet. *IEEE International conference on Programmable Control and Automation Technology*, pages 1– 10, 1988.

[20] M.O. Cordier and L. Rozé. Diagnosing discrete-event systems : extending the "diagnoser approach" to deal with telecommunication networks. *Journal on Discrete Event Dynamic Systems*, 12(2):43 – 81, 2002.

[21] M. Courvoisier, M.Combacau, and A. de Bonneval. Control and monitoring of large discrete event systems: a generic approach. *In Proc. of ISIE 93*, pages 571–576, 1993.

[22] G. Davis. *Introduction to Packaging Machines*. Arlington, 1997.

[23] dSPACE GmbH. dspace prototyping systems. http://www.dspaceinc.com.

[24] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. Optimal discrete controller synthesis for modeling fault-tolerant distributed systems. *Proceedings of the 1st IFAC Workshop on Dependable Control of Discrete Systems*, 2007.

[25] D. Orive E. Estévez, M. Marcos. Automatic generation of plc automation projects from component-based models. *Int J Adv Manuf Technol*, pages 527–540, 2007.

[26] M. Marcos E. Estévez. An approach to use model driven design in industrial automation. *13 th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 62–69, 2008.

[27] E. Faldella, A. Paoli, M. Sartini, and A. Tilli. Hierarchical supervision systems in industrial automation: a design procedure based on the generalized actuator concept. *Proceedings of 17th IFAC WC*, pages 69–76, 2008.

[28] E. Faldella, A. Paoli, A. Tilli, M. Sartini, and D. Guidi. Architectural design patterns for logic control of manufacturing systems: the generalized device. *XXII International Symposium on Information, Communication and Automation Technologies*, 2009.

[29] L. Ferrarini, R. Brusa, and C. Veber. A pragmatic approach to fault diagnosis in hydraulic circuits for automated machining: a case study. *4th IEEE Conference on Automation Science and Engineering*, 2008.

[30] L. Ferrarini, C. Veber, and G. Fogliazza. Iec 61499 implementation of a modular control model for manufacturing system. *IEEE International conference on Emerging Technologies and Factory Automation*, 1:7 – 13, 2005.

[31] L. Ferrarini, C. Veber, and V. Schirò. A modular modelling and implementation of automation functions for flexible manufacturing systems. *ANIPLA international Congress 50¡ Anniversary 1956-2006, Methodologies for Emerging Technologies in Auotomation*, 2006.

[32] FESTO-Didactic. *Assembly station manual*, 2003.

[33] FESTO-Didactic. *Distribution station manual*, 2003.

[34] FESTO-Didactic. *Processing station manual*, 2003.

[35] FESTO-Didactic. *Testing station manual*, 2003.

[36] E. Gamma. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[37] GAMP. Good automated manufacturing practice. www.ispe.org/gamp/.

[38] S. Genc and S. Lafortune. Distributed diagnosis of place-bordered petri nets. *IEEE Transactions on Automation Science and Engineering*, 4(2):206 –T 219, 2007.

[39] A.L. Buczak H. Darabi, M.A. Jafari. A control switching theory for supervisory control of discrete event systems. *IEEE Transactions on Robotics and Automation*, 19(1):131 – 137, 2003.

[40] N. Ben Hadj-Alouane, S. Lafortune, and F. Lin. Centralized and distributed algorithms for on-line synthesis of maximal control policies under partial observation. *Discrete Event Dynamic Systems: Theory and Applications*, 6(4):379 – 427, 1996.

[41] D. Harel. Statecharts: a visual formalism for complex systems. *Science of computer programming*, pages 231–274, 1987.

[42] D. Harel. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill, 1998.

[43] A. Hellgren, B. Lennartson, and M. Fabian. Modelling and plc-based implementation of modular supervisory control discrete event systems. *Proceedings of 6th Int. Workshop on Discrete Event Systems*, 2002.

[44] Standard IEC. *IEC 60848 Grafcet specification language for sequential function charts*. The Institution of Electrical Engineering, 2002.

[45] Standard IEC. *Programable controllers - Part 3: Programming language.* The Institution of Electrical Engineering, 2003.

[46] Standard IEC. *61499-1 Function blocks - Part 1: Architecture*. The Institution of Electrical Engineering, 2005.

[47] Standard IEC. *61499-2 Function blocks Ð Part 2: Software tools requirements*. The Institution of Electrical Engineering, 2005.

[48] M. V. Iordache and P. J. Antsaklis. Resilience to failure and reconfigurations in the supervision based on place invariants. *Proceedings of the 2004 American Control Conference*, 2004.

[49] ISA. *Batch Control Part 1: Models and Terminology*. ISA, 1995.

[50] ISA. *Batch Control Part 2: Data Structures and Guidelines for Languages*. ISA, 2001.

[51] ISPE. International society of pharmaceutical engineering. http://www.ispe.org.

[52] S. Jiang and R. Kumar. Diagnosis of repeated failures for discrete event systems with linear-time temporal logic specifications. *IEEE Transactions on Automation Science and Engineering*, 3(1):47 – 59, 2006.

[53] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall International, 1986.

[54] Kleanthis C. Thramboulidis. Using uml in control and automation: A model driven approach. *INDIN '04. 2nd IEEE International Conference*, pages 587–593, 24-26 June 2004.

[55] H. Koepetz. *Real-time systems: design principles for distributed embedded applications*. Real-time systems. Kluwer academic publishers, London, 1997.

[56] R. Leduc, B. Brandin, M. Lawford, and W. M. Wonham. Hierarchical interface-based supervisory control, part i: Serial case. *IEEE Transactions on Automatic Control*, 50(9):1322 –1335, 2005.

[57] R. Leduc, M. Lawford, B. Brandin, and W. M. Wonham. Hierarchical interface-based supervisory control, part ii: Parale case. *IEEE Transactions on Automatic Control*, 50(9):1336 –1348, 2005.

[58] R. Leduc, M. Lawford, and P. Dai. Hierarchical interface-based supervisory control of a flexible manufacturing system. *IEEE Transactions on Control Systems Technology*, 14(4):654 –668, 2006.

[59] R. W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3.* IEE Control Engineering Series, 1998.

[60] R. W. Lewis. *Modelling control system using IEC 61499*. The Institution of Electrical Engineering, 2001.

[61] J. Lunze. State observation and diagnosis of discrete-event systems described by stochastic automata. *Journal on Discrete Event Dynamic Systems*, 11(4):319 – 369, 2001.

[62] J. M. Machado, B. Denis, J. J. Lesage, J.M. Faure, and C. L. Ferreira da Silva. Logic controllers dependability verification using a plant model. *Proceedings of 3th IFAC Workshop on Discrete-Event System Design*, 2006.

[63] MathWork. Real time workshop. http://www.mathworks.com/products/rtw.

[64] H. E. Merrit. *Hydraulic control systems*. John Wiley & Sons, Inc. New York-London-Sydney, 1967.

[65] S. Moreno and E. Peulot. *Le GEMMA: modes de marches et d'arrest, Grafcet de coordination des taches, Conceptions des Systemes Automatises de Production surs.* Editions Casteillal, 2002.

[66] O.M.G. Uml, v. 1.4, omg specification, 2001. Document N. formal/2001-09-67, 2001, www.omg.org/uml.

[67] O.M.G. Uml, v. 2.0, omg request for proposal, 2003. Document N. ad/2003-03-02, 2003, www.omg.org/uml.

[68] A. Paoli and S. Lafortune. Safe diagnosability for fault tolerant supervision of discrete event systems. *Automatica*, 41(8):1335 – 1347, 2005.

[69] A. Paoli and S. Lafortune. Diagnosability analysis of a class of hierarchical state machines. *Discrete Event Dynamic Systems: Theory and Applications*, 18(3):385–413, 2008.

[70] A. Paoli, M. Sartini, and S. Lafortune. A fault tolerant architecture for supervisory control of discrete event systems. *Proceedings of 17th IFAC WC*, pages 6542–6547, 2008.

[71] A. Paoli, M. Sartini, E. Morganti, and C. Bonivento. Genetic algorithm for the optimization of packiing line. *ACD Workshop on Advanced Control and Diagnosis*, 2007.

[72] A. Paoli, M. Sartini, and A. Tilli. Rapid prototyping of logic control in industrial automation exploiting the generalized actuator concept. *13 th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 82–89, 2008.

[73] R.J. Patton, P.M. Frank, and R.N. Clark. *Issues of fault diagnosis for dynamical systems*. Springer-Verlag, 2000.

[74] A. Philippot, M. Sayed-Mouchaweh, and V. Carre-Menetrier. Distributed modeling approach of discrete manufacturing systems by parts of plant. *Proceedings of the 10th European Control Conference*, 2009.

[75] A. Pnueli. The temporal logic of programs. *In Proc. 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[76] J.M. Roussel and A. Giua. Designing dependable logic controllers using the supervisory control theory. *Proceedings of the 16th IFAC World Congress*, 2005.

[77] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 2004.

[78] M. Sampath, R. Sangupta, and S. Lafortune. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555 – 1575, 1995.

[79] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105 – 124, 1996.

[80] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105 – 124, 1996.

[81] M. Sayed-Mouchaweh, A. Philippot, V. Carre-Menetrier, and B. Riera. Fault diagnosis of discrete event systems using components fault-free models. *Proceedings of the 20th International Workshop on Principles of Diagnosis*, 2009.

[82] B. Selic. The real-time uml standard: definition and application. *IEEE International conference on Design. Automation and Test*, pages 770 – 772, 2002.

[83] B. Selic, G. Gullekson, and P. Ward. *Real-tim object -oriented modeling*. John Willey & Sons, 1994.

[84] J.M. Spivey. *Introducing Z: a specification language and its Formal semantics. Statecharts: a visual formalism for complex systems.* Cambridge university press, 1988.

[85] J. Sifakis T. A. Henzinger. The discipline of embedded systems design. *IEEE Computer*, 40(10):32 –40, 2007.

[86] K. C. Thramboulidis. Towards an engineering tool for implementing reusable distributed control system. *ACM SIGSOFT Software Engineering Notes*, 28(5), 2003.

[87] K. C. Thramboulidis. Model integrated mechatronics-towards a new paradigm in the development of manufacturing systems. *IEEE Transaction on Industrial Informatics*, 1:54 – 61, 2005.

[88] K. C. Thramboulidis and C. S. Tranoris. Developing a case tool for distributed control application. *The Int. Journal of Advanced manufacturing*, 28(1-2), 2004.

[89] A. Tilli and A. Paoli. Rule-based composable modelling of industrial automation automata under nominal and faulty conditions. *Proceedings of the 7th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes*, 2009.

[90] A. Tilli, A. Paoli, M. Sartini, C. Bonivento, and D. Guidi. Hierarchical and cooperative approaches to logic control design in industrial automation. *14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 82–89, 2009.

[91] T. Tolio. *Design of flexible production system*. Springer, 2009.

[92] V. Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. Instrumentation Society of America, USA, July 2006.

[93] V. Vyatkin and X. Cai. Design and implementation of a prototype control system according to iec 61499. *IEEE International conference on Emerging Technologies and Factory Automation*, 2(269-276), 2003.

[94] V. Vyatkin and H. M. Hanisch. Formal modeling and verification in the software engineering framework of iec 61499: a way to self-verifying systems. *IEEE International conference on Emerging Technologies and Factory Automation*, 2(113–118), 2001.

[95] Q. Wen, R. Kumar, J. Huang, and H. Liu. Fault-tolerant supervisory control of discrete event systems: formalisation and existence results. *Proceedings of the 1st IFAC Workshop on Dependable Control of Discrete Systems*, 2007.

[96] Q. Wen, R. Kumar, J. Huang, and H. Liu. Weakly fault-tolerant supervisory control of discrete event systems. *Proceedings of the 2007 American Control Conference*, 2007.

[97] Q. Wen, R. Kumar, J. Huang, and H. Liu. A framework for fault-tolerant control of discrete event systems. *IEEE Transactions on Automatic Control*, 53(8):1839 – 1849, 2008.

[98] W. M. Wonham. Notes on control of discrete event systems. ECE 1636F/1637S 2002-2003. Systems Control Group, Dept. of ECE, University of Toronto, URL: www.control.utoronto.ca/people/profs/wonham/wonham.html.

[99] T.-S. Yoo and S. Lafortune. Solvability of centralized supervisory control under partial observation. *Discrete Event Dynamic Systems: Theory and Applications*, 16(4):527 – 553, 2006.

[100] Y. Zhang and J. Jiang. Integrated active fault-tolerant control using imm approach. *IEEE Transactions on Aerospace. and Electronic Systems*, 37(4):1221 – 1235, 2001.

Index

Curriculum vitae

Matteo Sartini was born in Rimini (Italy) on 15 December 1978. He got his diploma degree at Istituto Tecnico Industriale Statale Leonardo Da Vinci in Rimini with the mark of 60/60. He took his master degree in Computer Science Engineeering at University of Bologna in March 2005 with a master thesis titled "Modellistica mediante sistemi ibridi: applicazione al sistema frizione driveline in ambiente automotive - Hybrid systems modeling: application to clutch driveline in automotive systems". On January 2006 he obtained his professional engeneering degree.
In July 2005 he won a research grant supported by the University of Bologna and Emilia-Romagna regional Council for the project titled "Diagnosis and control for fault tolerant automation systems" under the supervision of Prof. Claudio Bonivento.
On January 2006 he began his PhD (XXII cycle) and his research activity within D.E.I.S. (Dipartimento di elettronica, Informatica e Sistemistica - Department of Electronics, Computer Science and Systems) under the supervision of Prof. Claudio Bonivento.
In January 2010 he won a research grant supported European Artemis Joint Undertaking funded project CESAR: *Cost-Efficient methods and processes for SAfety Relevant embedded systems* under the supervision of Dr. Andrea Paoli.
His main research interests are: Industrial automation software architectures, Discrete Event Systems, and fault tolerant control architectures in order to obtain safety and tolerance to faults.