UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROGRAMA DE DOCTORAT:

AUTOMÀTICA, ROBÒTICA I VISIÓ

DOCTORAL THESIS

FAST: a Fault Detection and Identification Software Tool

JORDI DUATIS JUÁREZ

DIRECTOR: CECILIO ANGULO BAHÓN

December 2, 2015

Declaration of Authorship

I, Jordi DUATIS, declare that this thesis titled, 'FAST: a Fault Detection and Identification Software Tool' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- No part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"'Failure is not an option' is a sentence originated during the discussions to prepare a film about the space program from Sputnik through the moon missions. During an interview to Jerry Bostick, NASA Flight Controller, the writers asked 'Weren't there times when everybody, or at least a few people, just panicked?' His answer was 'No, when bad things happened, we just calmly laid out all the options, and failure was not one of them. We never panicked, and we never gave up on finding a solution.'"

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

ESAII
Departament d'Enginyeria de Sistemes, Automàtica i Informàtica Industrial
Programa: Control, Visió, Robòtica (CVR)

Doctor by the UPC

FAST: a Fault Detection and Identification Software Tool

by Jordi DUATIS

The aim of this work is to improve the reliability and safety of complex critical control systems by contributing to the systematic application of fault diagnosis. In order to ease the utilization of fault detection and isolation (FDI) tools in the industry, a systematic approach is required to allow the process engineers to analyze a system from this perspective. In this way, it should be possible to analyze this system to find if it provides the required fault diagnosis and redundancy according to the process criticality. In addition, it should be possible to evaluate what-if scenarios by slightly modifying the process (f.i. adding sensors or changing their placement) and evaluating the impact in terms of the fault diagnosis and redundancy possibilities.

Hence, this work proposes an approach to analyze a process from the FDI perspective and for this purpose provides the tool *FAST* which covers from the analysis and design phase until the final FDI supervisor implementation in a real process. To synthesize the process information, a very simple format has been defined based on XML. This format provides the needed information to systematically perform the Structural Analysis of that process. Any process can be analyzed, the only restriction is that the models of the process components need to be available in the *FAST* tool. The processes are described in *FAST* in terms of process variables, components and relations and the tool performs the structural analysis of the process obtaining: (i) the structural matrix, (ii) the perfect matching, (iii) the analytical redundancy relations (if any) and (iv) the fault signature matrix.

To aid in the analysis process, *FAST* can operate stand alone in simulation mode allowing the process engineer to evaluate the faults, its detectability and implement changes in the process components and topology to improve the diagnosis and redundancy capabilities. On the other hand, *FAST* can operate on-line connected to the process plant through an OPC interface. The OPC interface enables the possibility to connect to almost any process which features a SCADA system for supervisory control. When running in on-line mode, the process is monitored by a software agent known as the Supervisor Agent.

FAST has also the capability of implementing distributed FDI using its multi-agent architecture. The tool is able to partition complex industrial processes into subsystems, identify which process variables need to be shared by each subsystem and instantiate a Supervision Agent for each of the partitioned subsystems. The Supervision Agents once instantiated will start diagnosing their local components and handle the requests to provide the variable values which FAST has identified as shared with other agents to support the distributed FDI process.

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

ESAII
Departament d'Enginyeria de Sistemes, Automàtica i Informàtica Industrial
Programa: Control, Visió, Robòtica (CVR)

Doctor by the UPC

FAST: a Fault Detection and Identification Software Tool

by Jordi DUATIS

Per tal de facilitar la utilització d'eines per la detecció i identificació de fallades (FDI) en la indústria, es requereix un enfocament sistemàtic per permetre als enginyers de processos analitzar un sistema des d'aquesta perspectiva. D'aquesta forma, hauria de ser possible analitzar aquest sistema per determinar si proporciona el diagnosi de fallades i la redundància d'acord amb la seva criticitat. A més, hauria de ser possible avaluar escenaris de casos modificant lleugerament el procés (per exemple afegint sensors o canviant la seva localització) i avaluant l'impacte en quant a les possibilitats de diagnosi de fallades i redundància.

Per tant, aquest projecte proposa un enfocament per analitzar un procés des de la perspectiva FDI i per tal d'implementar-ho proporciona l'eina FAST la qual cobreix des de la fase d'anàlisi i disseny fins a la implementació final d'un supervisor FDI en un procés real. Per sintetitzar la informació del procés s'ha definit un format simple basat en XML. Aquest format proporciona la informació necessària per realitzar de forma sistemàtica l'Anàlisi Estructural del procés. Qualsevol procés pot ser analitzat, només hi ha la restricció de que els models dels components han d'estar disponibles en l'eina FAST. Els processos es descriuen en termes de variables de procés, components i relacions i l'eina realitza l'anàlisi estructural obtenint: (i) la matriu estructural, (ii) el *Perfect Matching*, (iii) les relacions de redundància analítica, si n'hi ha, i (iv) la matriu signatura de fallades.

Per ajudar durant el procés d'anàlisi, FAST pot operar aïlladament en mode de simulació permetent a l'enginyer de procés avaluar fallades, la seva detectabilitat i implementar canvis en els components del procés i la topologia per tal de millorar les capacitats de diagnosi i redundància. Per altra banda, FAST pot operar en línia connectat al procés de la planta per mitjà d'una interfície OPC. La interfície OPC permet la possibilitat de connectar gairebé a qualsevol procés que inclogui un sistema SCADA per la seva supervisió. Quan funciona en mode en línia, el procés està monitoritzat per un agent software anomenat l'Agent Supervisor.

Addicionalment, FAST té la capacitat d'implementar FDI de forma distribuïda utilitzant la seva arquitectura multi-agent. L'eina permet dividir sistemes industrials complexes en subsistemes, identificar quines variables de procés han de ser compartides per cada subsistema i generar una instància d'Agent Supervisor per cadascun dels subsistemes identificats. Els Agents Supervisor un cop activats, començaran diagnosticant els components locals i despatxant les peticions de valors per les variables que FAST ha identificat com compartides amb altres agents, per tal d'implementar el procés FDI de forma distribuïda.

Acknowledgements

This thesis is the result of a long way. The first ideas started when I was introduced to the MELiSSA project in 2002, which is a project from the Europan Space Agency (ESA) with the eventual objective of providing the full recycling capability of human wastes in a closed environment such as a planetary base or a long journey space ship. This project impacted me mainly because from my perspective it implies a new step in the human evolution, if finally, the humankind is able to become independent from the mother Earth and travel beyond the stars being self-sustainable. Obviously, apart from this philosophical dissertation, there is a more practical result, which comes from the fact that if we are able to recycle almost the 100% of the human wastes in a closed environment, it will benefit the preservation of the Earth environment.

Thus my first acknowledgment, in chronological order, is to Joan Mas and Miquel Pastor, who authorized my involvement in the MELiSSA project in those early dates. Also I acknowledge the contribution of Christophe Lasseur who from ESA valued the work we performed in the project and allowed us a continuous contribution to it until now.

I provide also my sincere acknowledgment to Cecilio Angulo and Vicenç Puig who supported me in this long way, providing interesting points of view and directing my work to an achievable objective, which has been an arduous task.

Contents

List of Figures

List of Tables

List of Abbreviations

FAST	**F**DI **A**gent **S**oftware **T**ool.
FDI	**F**ault **D**etection and **I**dentification.
OLE	**O**ject **L**inking and **E**mbedding. Old MS Windows technology to intercommunicate computer processes.
FMEA	**F**ailure **M**ode and **E**ffects **A**nalysis.
FSM	**F**ault **S**ignature **M**atrix.
OPC	**O**LE for **P**rocess **C**ontrol.
PDL	**P**rocess **D**efinition fi**L**e.
PLC	**P**rogrammable **L**ogic **C**ontroller.
PM	**P**erfect **M**atching.
P&ID	**P**rocess and **I**nstrumentation **D**iagram.
SCADA	**S**upervisory **C**ontrol **A**nd **D**ata **A**cquitision.
SM	**S**tructural **M**atrix.

List of Symbols

a	arc in a bipartite graph
$\mathcal{C}$	Set of components
f	fault vector
r	component relation
$\mathcal{R}$	Set of component relations
$\mathcal{V}$	Set of process variables
x	process variable
z	residual
$\mathcal{Z}$	Set of residuals

Definitions

Fault	A fault is any not permitted deviation of at least one characteristic property (feature) of the system from the acceptable, usual, standard conditions.
Failure	A failure is a permanent interruption of a system's ability to perform a required function under specified operating conditions.
Malfunction	A malfunction is an intermittent irregularity in the fulfillment of a system's desired function.
Reliability	Ability of a system to perform a required function under stated conditions, within a given scope, during a given period of time.
Safety	Ability of a system not to cause danger to persons or equipment or the environment.
Availability	Probability that a system or equipment will operate satisfactorily and effectively at any period of time.
Dependability	A form of availability that has the property of always being available when required (and not at any time). It is the degree to which a system is operable and capable of performing its required function at any randomly chosen time during its specific operating time, provided that the system is available at the start of the period.
Integrity	Safety integrity is the probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a period of time.

1

[1] Definitions as used in the text and reproduced from [Isermann, 2006]

Dedicated to my wife, sons, parents and brothers. Who allays believed on me and supported me during the long journey. All efforts are sooner or later rewarded.

Chapter 1

Introduction

1.1 Motivation

The European Space Agency has been working since more than 25 years in a concept of a bio-regenerative Life Support System. The project known as MELiSSA (Micro-Ecological Life Support System Alternative) has as objective to develop the research in artificial ecosystems and recycling technologies and in the last term to reach a bio-regenerative life support system for long space journeys and for permanent planetary bases [Binot, Tamponnet, and Lasseur, 1994]. The project is based on reproducing the ecosystem which can be found in a lake, from the lower deep-water layers, with few light and a lot of organic compounds up to the upper layers with a lot of light and oxygen with the objective of recycling human wastes in fresh water, oxygen and food. The layers are reproduced separately in bio-reactors, with strict control of the environmental conditions (temperature, pressure, pH, etc.). An experimental plant in form of a pilot plant is installed in the Chemical Engineering Department of the *Universitat Autònoma de Barcelona, UAB*. The MELiSSA Pilot Plant - Claude Chipaux Laboratory is composed of a set of bio-reactors with specific microorganism cultures which in a controlled and isolated form reproduce the bio-degradation processes emulating the lake ecosystem. The objective of the pilot plant is to characterize the bio-reactors and implement experiments to validate the models and the associated basic research.

The first contribution of the author was in 2002 as the responsible to design a new Control System Architecture for the MELiSSA Pilot Plant with the objective of covering current and future requirements, scalable, with a high reliability and availability and easy to maintain. The implementation of this project followed the classical approach, studying the process requirements, the currently available technologies and applying mature standards. However, during the implementation it was clear that the process was very complex and that even at pilot plant scale, a minor fault could cause that an experiment which lasted months becomes useless. The current implementation of fault detection is the traditional signal analysis based on thresholds, generating alarms which can be monitored in the SCADA system. Although this method is useful to avoid catastrophic faults, it is clear that faults are detected too late and the recovery process takes too long. For instance the loss of the biomass of a bio-reactor can take months to recover.

The MELiSSA project is exciting in the sense that implies the possibility of jumping to the stars with lack of dependence of supplies from Earth and at the same time contributes to the evolution of the recycling technologies which can be used on-ground (a Water Recovery Unit experiment is being used in the Concordia Antarctic base [Lasseur et al., 2004]) . However, it is clear that a life support system is a critical part of the mission with direct dependence on the crew safety and needs to operate transparently and

continuously. Therefore, fault diagnosis techniques will be even more relevant for a real space implementation.

Nevertheless, after starting the development of this work the author quickly realized that the application and benefits are widely applicable. From industrial plants to complex systems, any process with a certain level of complexity involving sensors and actuators and control loops, should apply a systematic fault diagnosis methodology as part of the engineering design process. In the space sector, any system is critical, in the sense of the lack of repair possibilities and the high development and operation costs. In space projects, the fault analysis is an essential part of the design process, trying to identify each fault possibility for each single component and the effects of fault propagation. However, it is clear that it is impossible to identify all potential faults and implement a totally fault tolerant system just covering the faults identified at design level, examples are widely known even in this mission critical sector. Therefore fault diagnosis during system operation (i.e. Active Fault Diagnosis) should be also considered systematically and incorporated into all on-board FDIR *Fault Detection, Identification and Recovery* subsystems.

There is a long way in developing fault diagnosis research and there are several methods already proposed. However, the implementation in the industry is not systematic and each sector has its own recipes. The research in fault diagnosis methods is quite transversal, in the sense that can be applied to any technology or discipline. The well-known FMEA *Failure Modes and Effects Analysis* technique offers a systematic approach to design systems with a high level of reliability. For critical processes, the fault detection and fault tolerance approach is via implementing hardware redundancy, duplicating sensors, actuators and even complete subsystems with the objective of avoiding that single faults cause a total mission loss or a process emergency stop. However, the advantage that model-based fault diagnosis and the analytical redundancy can provide to the on-line fault diagnosis should be also considered, specially for industrial processes where the cost of having hardware redundancy is difficult to assume. But even for the critical systems with a high degree of hardware redundancy, a systematic approach for active fault detection and isolation during the system operation is of primary importance since it provides valuable information which can be used to generate a more reliable fault diagnosis and therefore better possibilities of implementing effective active fault tolerance.

1.2 Previous Work

The Structural Analysis applied to FDI was proposed by [Declerck and Staroswiecki, 1991], [Cassar, Staroswiecki, and Declerck, 1994] and [Staroswiecki, Attouche, and Assas, 1999], it has been refined and evolved in several posterior works [Izadi-Zamanabadi, 1999] and it is also extensively described in [Blanke et al., 2006].

The objective of having a tool to aid the FDI process design has been approached by other research groups and as a result there are two relatively known tools which have been developed; the SATOOL [Blanke and Lorentzen, 2006] and the Bond Graphs tool SYMBOLS-2000 [Bouamama et al., 2005]. SATOOL was able to implement up to the perfect matching and the parity relations by symbolic transformations. The tool is implemented in MATLAB. The Bond Graphs tool is able to define a process by graphically connecting component models, derive the bond-graph representation and simulate the process.

Distributed FDI has been also subject of research in various projects. In [Llanos Rodríguez, 2008], the time constraints are studied in detail and in [Ferrari, 2008] decentralized and distributed fault diagnosis for discrete and continuous time systems are analyzed. In [Belkacem and Bouamama, 2015], it was presented a possible identification of subsystem candidates for a distributed FDI implementation based on the Fault Signature Matrix, in a similar way as the proposed in this work. A decentralised FDI implementation for water management network is also discussed in [Puig and Ocampo-Martínez, 2015]. Distributed FDI is also a matter of research in swarm robotics and other coordinated organizations of autonomous systems.

A work was proposed involving software agents in the FDI process as part of the research project MAGIC [Köppen-seliger, Ding, and Frank, 2002] and [Ploix, Gentil, and Lesecq, 2003]. The work proposed an architecture defining a hierarchical structure distributed in several layers each one implementing a specific function. A complete architecture for active fault diagnosis applied to bio-regenerative life-support systems was proposed in [Duatis et al., 2008].

1.3 Objectives and Contributions

The objectives in this thesis are twofold: From one side to propose a systematic approach to active fault diagnosis analysis of a system which could be easily applied to any process and which could be used to improve the system design from the FDI perspective. On the other hand, to provide an architecture suitable to implement on-line distributed fault diagnosis. In this sense, the implementation of on-line fault diagnosis through software agents becomes a natural approach after analyzing the capabilities of this relatively new software engineering paradigm.

The first contribution of this work is the implementation of a tool to perform the Structural Analysis of a process in a simple and systematic way. The Structural Analysis is a powerful tool to obtain the active fault diagnosis capabilities of a process. From a preliminary design, just by providing the process components information, it is possible to analyze which are the fault detection and isolation inherent possibilities and identify the improvements to the process increasing this capability. The structural analysis implemented in *FAST* is detailed in Chapter 4. The tool also allows the process engineer to simulate the fault injection into the process and evaluate the effects.

The second contribution is the introduction of the software agents in the implementation of the on-line fault diagnosis. In addition, with special relevance for industrial processes, an OPC interface has been incorporated to these software components enabling their direct interaction with a real process and obtain in real-time the measured variable values and feed the process with diagnostic results. The OPC (OLE for Process Control) is a mature technology used for SCADA systems to access to the supervision information of industrial processes. This standard is managed by the OPC Foundation [1]. The *FAST* tool implements the Supervisor Agent responsible of the on-line fault diagnosis process. This agent is able, via the OPC interface, to instantiate a Knowledge Base with the process information, updated it in real-time and provide feedback to the process and/or to the SCADA supervision system. The architecture of the Supervisor Agent is explained in Section 3.4.

The third contribution is the possibility of implementing distributed FDI by using the software agents taking the advantage of their capability to exchange information

[1] More information in OPC Foundation, http://opcfoundation.org (visited 07/10/2015).

and be self-organized in a network as a multi-agent system. *FAST* provides the functionality of analyzing a process and detecting possible subsystem candidates to be supervised in a distributed way based on the concept of components coupling. Each subsystem will provide its local information to a local Supervisor Agent, which will use this information to implement the local diagnosis process. However, the Supervisor Agent working in cooperation with the neighbor agents will be also able to consolidate information from other connected subsystems and extend the fault detection capabilities by using this information. The distributed FDI is illustrated in Chapter 5.

1.4 Structure of the Thesis

In this Chapter 1, the author presents the motivations which resulted in the embarking into this project, a brief indication of the previous research on the matter, the objectives and contributions of this work and the organization followed to expose all the themes. In Chapter 2, the theory in which all the development of this work is founded is summarized. This project is based on the previous research in Fault Diagnosis and Identification discipline, specially on model based fault diagnosis and the structural analysis method. On the other hand, the software agents paradigm is visited in order to offer a sufficient background to understand the proposed architecture to implement the real-time and distributed FDI. Chapter 3 details the *FAST* tool architecture. This tool is composed of two different components, the *FAST Simulator* and the *FAST Supervisor Agent*. The simulation component is the consequence of the research in a method to facilitate the FDI analysis of a process in a systematic way and the Supervisor Agent is the natural implementation after analyzing the requirements to implement on-line real time fault diagnosis in a distributed organization. The Chapter 4 is devoted to the detailed exposition of the Structural Analysis implementation in the *FAST* tool. The algorithms used are explained and an application example is provided. The distributed FDI implementation is discussed in the Chapter 5, where the algorithm to determine subsystem candidates to be supervised in a distributed way is exposed. In this chapter, the implementation details are explained and an application example of a water distribution system is used to illustrate this implementation. Finally, in Chapter 6 some conclusions are presented and a list of recommendations for future work are discussed. The appendixes are the precursors of a user manual to ease the utilization of *FAST*, with tool user instructions indicated in the Appendix A and how to add additional component models to the tool explained in the Appendix B.

Chapter 2

Fundamentals and State of the Art

2.1 Fault Diagnosis and Identification (FDI)

2.1.1 Introduction to FDI

The FDI research has been developed to increase the reliability of critical processes by providing systematic techniques to design fault diagnosis systems and as a preliminary mandatory step for fault tolerant control implementation. Furthermore, FDI is able to provide a valuable assistance to plant operators when there is any fault in the system by identifying the component causing the fault and improving the maintainability of the system. With the proliferation of the automatic control and its wide implementation in many critical services nowadays, the continuous operation is a compulsory requirement implying human safety and material risks. Eventually the objective of FDI is to detect any fault in the process and identify uniquely which is the process component which is causing the fault in a systematic way and in a limited (short) time. Therefore, we can identify three phases in the FDI process:

- Fault detection: identification of the existence of a fault and the time of its occurrence.
- Fault isolation: identification of the component originating the fault.
- Fault identification and estimation: identification of the fault mode and estimation of its magnitude.

There are many types of faults which can occur in an automated process, just to mention a few:

- The electrical fault of a sensor, leading to provide a wrong value or no value at all.
- The leakage of a valve, a tank or a pipe.
- The clogging of some filter, valve or pipe.
- The data connection failure with some control devices
- A change in the environmental conditions of the process which causes a perturbation of the process. For instance, the stop of the pressurized air supply to actuate pneumatic valves or the variation of the environmental humidity or temperature to an uncontrollable levels.
- A device which by its use has degraded its performance and is not reaching the adequate level to maintain the objectives of the process.

- A wrong operation of some device by a plant operator which causes some process to deviate its nominal performance.
- A design problem not detected which causes some continuous or intermittent deviation of the nominal behavior of the plant. For instance, a bug in the PLC control algorithms, or the bad dimensioning of an installed device.

FDI is also the first step to apply fault tolerant control, as to design a fault tolerant system first it is needed to study the possible faults, its detection capability, and how to accommodate them in the control of the process.

2.1.2 Fault Detection in Traditional Control Systems

In traditional control systems, the faults are detected using signal analysis methods such as *limit-checking* and/or spectral signal analysis [Isermann, 2006]. Isermann defines a *Fault* as "any not permitted deviation of at least one system parameter or property which deviates it from an acceptable situation".

A traditional control system can be represented by the Figure 2.1.

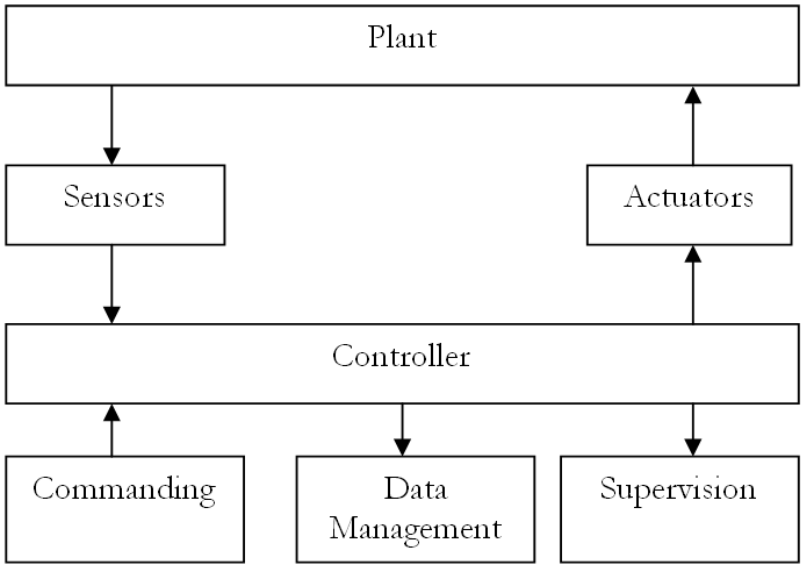


FIGURE 2.1: Traditional control system architecture

In direct interaction with the plant, we have the sensors and the actuators. The sensors will read plant outputs which will feed the controller and provide information about the plant process behavior. Actuators will provide means to influence the plant process and control its behavior, providing the optimal conditions for the process to achieve its objectives.

The controller will acquire the data provided by the sensors and will process it via the control algorithms used to bring the process variables to the defined set-points.

Usually, there is a Data Management System which archives the information provided by the sensors and by the controller. With these data, it will be possible generate trend graphs and check the behavior of the system during specific time intervals.

The Commanding will be the part of the Control System to implement maintenance and troubleshooting. It can be used to operate manually some actuators, change the mode of operation of the controller (from automatic to manual, or to OFF, for instance).

The Supervision system will be used to monitor the process behavior in real-time. The information provided by the controller is displayed graphically in a computer software and any deviation of a process variable is displayed as an alarm. The Supervision system can incorporate the Alarm Management system, which will contain the acceptable limits of the process variables when the system is working in nominal mode. If any process variable goes beyond these limits for a certain time, the Alarm Management system will generate an alarm. This alarm is archived and normally represented in a graphical way in the Supervision software.

The Alarm Management system also provides information to perform maintenance interventions as some alarms can be caused by components which are not working properly or have degraded its function to a level to which is affecting the process.

Traditionally, the supervision of the systems is performed by humans, which for critical processes (such as a nuclear reactor) requires continuous monitoring 24 hours a day, every day. The supervision operators monitor visually that the plant processes are performed according to the expectations and no critical alarms are generated. When a critical alarm is detected by a human supervisor, he/she will apply a known procedure, by following a maintenance manual or by advise of an expert. The procedure can mitigate the problem detected by the alarm changing the operation mode of the process or a maintenance intervention interrupting part of the process to change or repair some components.

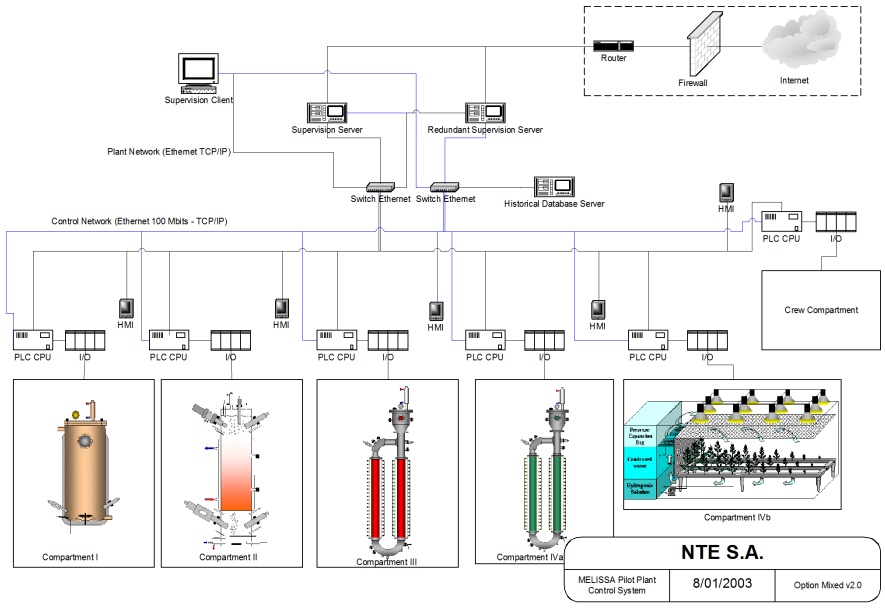


FIGURE 2.2: MELiSSA Control System Architecture

2.1.3 FDI Detection Methods

Nowadays it is possible to identify three groups of detection methods:

- Methods based on the signal analysis.

- Methods based on models.
- Methods based on the process knowledge

In addition, the different FDI techniques can also be classified in the detection and diagnostic phases. The objective of the detection phase is to detect if there is a fault in the system while the objective of the diagnostic phase is to identify the fault, classify it and to determine the originating cause.

2.1.4 Model Based Fault Detection

The key of Model based fault detection is to obtain the measures of the inputs and outputs of the process, feed it into a process model and compare the real process behavior with the model.

A process can be represented by:

$$\begin{aligned} \dot{x}(t) &= g(x(t), u(t), \theta) \\ y(t) &= h(x(t), u(t), \theta) \end{aligned} \tag{2.1}$$

where:

- $x \in \Re^{nx}$, $u \in \Re^{nu}$ and $y \in \Re^{ny}$ are the state space , the input and output vectors respectively.
- g and h are the state space functions and measure respectively.
- θ is the parameters vector of dimension p.

2.1.5 Detection Based on Quantitative Models

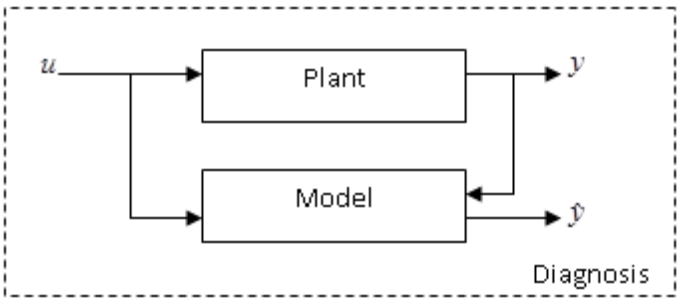


FIGURE 2.3: Model based fault detection

In the Figure 2.3, we can identify:

- $u = u_0, u_1, ..., u_k$ as the measured process inputs.
- $y = y_0, y_1, ..., y_l$ as the measured process outputs
- $\hat{y} = \hat{y}_0, \hat{y}_1, ..., \hat{y}_m$ as the model estimated outputs.

The principles of the model based fault detection are based on the calculation of the difference between the estimated outputs by the model and the measured outputs $y - \hat{y}$. Where the model is calculated from the measurement of the inputs and the outputs of the process.

Therefore, we define a *residual* as the result of the difference of the real output and the estimated output and is formulated as:

$$r(t) = y(t) - \hat{y}(t) \tag{2.2}$$

When the relation becomes more complex involving also the process inputs $u(t)$ and the parameters θ, it is known as *analytical redundant relation* (ARR) and can be represented as $\psi(t)$:

$$\psi(t) = f(y(t), u(t), \theta) \tag{2.3}$$

The most utilized formal methods to generate residuals by means of analytical models are:

- parity equations proposed by Gertler in [Gertler, 1991].
- observers proposed by Chen in [Chen, 1995].
- parameter estimation proposed by Isermann in [Isermann, 2005].

The most known techniques to generate analytical redundant relations are:

- parity space proposed by Chow in [Chow and Willsky, 1984].
- structural analysis proposed by Staroswiecki and Cassar in [Staroswiecki, Cassar, and Declerck, 2000].

Note that in lack of presence of a fault, the residual evaluation will approximate to 0 while if there is some fault, the residual should be different from 0. However, this is only in the ideal case, as the sensors noise, the plant perturbations and the errors in the models will make not possible to detect if there is a fault in the system just by determining if a residual is null or not. In order to be able to detect a fault, the residuals should be insensitive to the noise, perturbations and model errors being this the objective of designing *robust* residuals. The process of generating robust residuals can be performed during the residual generation phase or during the evaluation phase. When the robust residual design is applied during the residual generation phase, it is known as *active robustness* while when the robust residual design is applied during the residual evaluation phase it is known as *passive robustness*.

The passive robustness is basically to design properly the thresholds that the residuals should pass to determine that there is a fault in the system. These thresholds need to be defined to be insensitive to the noise and perturbations and taking into account the process operational modes applying adaptive techniques. The advantage of these techniques is that they can be adjusted to avoid the effect of uncertainty in the models. This is achieved by adjusting the thresholds in such that when the process is working properly no fault is reported. However, the definition of the thresholds in this way lead to the lack of detection of faults which cause only a small perturbation in the residuals.

The active robustness includes the uncertainty in the calculation of the residual and tries to decouple the perturbations from them. Several techniques have been proposed by [Gertler, 1998], [Chen and Zhang, 1991] or [Frank, 1994] among others.

This work is based on the generation of analytical redundant relations based on the Staroswiecki proposed structural analysis [Declerck and Staroswiecki, 1991], [Cassar, Staroswiecki, and Declerck, 1994], [Staroswiecki, Attouche, and Assas, 1999] and the application of passive robustness to the evaluation of the residuals as explained in Chapter 3.

2.1.6 Detection Based on Qualitative or Semi-qualitative Models

These techniques are used when it is not possible to obtain an accurate or complete model of the system, or merely the process does not need such precision. In this case, just by using qualitative models describing only the primary and substantial relations and ignoring relations which are not relevant or unknown. In this case, the relations will define system properties where instead of numerical values, are represented by a quality attribute (grows, decreases, maintains, positive, negative, etc.). Semi-qualitative models use characterized value sets, intervals or fuzzy sets. These models have the drawback that are imprecise and some times the knowledge of the fault model is required prior to the detection process design, but have the advantage that do not need the complete model of the process to implement fault detection.

2.1.7 Detection Based on Soft-computing Methods

When there is no model at all, it is still possible to implement fault detection specially in highly non-linear processes by using soft-computing techniques, such as:

- Neural networks
- Fuzzy logic
- Genetic algorithms
- Expert systems

Some techniques combine neural networks with fuzzy logic to take the advantage of the learning process of the neural networks and the qualitative reasoning provided by the fuzzy logic. The advantage of these techniques is that can be applied with very few knowledge of the model of the system, however they lack of a deterministic behavior and it is difficult to verify its function in all possible operation modes.

2.1.8 Fault Diagnosis

The fault diagnosis will consist of the fault isolation and identification processes. The fault isolation is the process to determine the cause of the fault while the identification is the process of quantifying the type, magnitude and severity of the fault. There are mainly two approaches to the fault diagnosis although most of the concepts are common [Puig et al., 2000]:

- the FDI approach, near to the control engineering discipline.
- the DX approach, near to the artificial intelligence techniques.

The DX approach is based on the implementation of model based approaches using logic reasoning techniques coming from the artificial intelligence community [Kleer, Mackworth, and Reiter, 1992, Reiter, 1987]. DX fault diagnosis has been initially developed for combinatory logic circuits, at later extended to dynamic systems. The diagnosis algorithms offer a sound and complete diagnosis able to cope with multiple faults. However, the consistency based DX algorithms are too complex to be operated on-line with dynamic systems. Recent results show that they can be used to generate a set of relations (possible conflicts) which can be evaluated on-line in a similar manner than ARR in FDI [Pulido and Gonzalez, 2004].

The FDI approach is based on an existing set of residuals which can be evaluated in real-time and generate a fault vector. This vector is compared with a fault matrix where for each process component there is a dependency relation between the component and a fault. Then, by comparing the fault vector with the fault matrix, it is possible for some faults identify the component originating the fault.

2.2 The Structural Analysis

The structural analysis is a technique proposed by Staroswiecki in [Staroswiecki and Declerck, 1989] which consists on performing an analysis of the structural properties of the model of a process $\mathcal{S}$, that is, considering process properties which are independent of the value of the parameters. In principle, the structural model definition of a process is independent of the nature of the relations involved (quantitative, qualitative, equations, rules, etc.) as only takes into account the link between the relations and the variables linked to the relation. Note that in this work we will indicate as relations the set of model properties which describe the behavior of a component. In [Blanke et al., 2006], the relations are named *constraints*. The structural analysis is based on the generation of a bi-partite graph between the relations and the variables of the model. These links are indeed independent of the representation of the relations and represent a qualitative, very low level and easy to obtain model of the system behavior. Although its simplicity, the structural model provides very valuable information such as:

- Identify such components which are (or not) monitored.
- Provide design approaches for analytical redundancy based residuals.
- Suggest alarm filtering strategies
- Identify those components whose failure can be tolerated through reconfiguration.

In the case that the relations are analytical and hold a specific set of properties, it will also be possible to obtain analytical redundancy relations which can be used to generate a set of residuals. From these residuals, it will be possible for some faults to implement the fault detection, isolation and identification phases. The component behavior will be described by a set of relations. Therefore, by establishing a link between the relations of a component participating in a residual and the residual, it will be possible to match non-zero residuals with faulty components.

2.2.1 The Structural Model

The structural model of a process $\mathcal{S}$ is generated as a bi-partite graph representing the links between a set of relations and a set of variables. The behavior of the system

can be described by the pair $(\mathcal{R}, \mathcal{V})$ being $\mathcal{R} = \{r_1, r_2, \cdots, r_n\}$ the set or relations and $\mathcal{V} = \{v_1, v_2, \cdots, v_m\}$ the set of variables and parameters. The set of relations can be of different forms (e.g. algebraic, differential equations, rules, etc.).

The relations must hold two properties in order to be considered for the analysis of the structural model:

- The relations $r_i \in \mathcal{R}$ are compatible. Meaning that the relation really describes the behavior of a component, hold a solution and are not contradictory.
- The relations $r_i \in \mathcal{R}$ are independent. That is, there is not the case where there is a set of relations which solutions are included into another set of relations, that is, being $\varphi(R_1)$ the solutions set corresponding to the relations in R_1 and $\varphi(R_2)$ another set of solutions corresponding to the relations in R_2, it is never the case that $\varphi(R_1) \subseteq \varphi(R_2)$.

Definition 1 *The bi-partite graph is formed by considering all the edges linking a relation with a process variable, that is, being $\mathcal{A}$ the set of edges relating the set of variables $\mathcal{V}$ to the set of relations $\mathcal{R}$: the edge $(r_i, v_j) \in \mathcal{A}$ if the variable v_j appears in the relation r_i.*

The bi-partite graph is formed by representing as bars the relations r_i and in circles the process variables v_j.

The example of the single tank system will be used along the Section to illustrate the structure analysis. This system is composed of the following set of relations representing a continuous time, continuous variable model:

- Tank $r_1 : \dot{h} = q_i(t) - q_o(t)$
- Input valve $r_2 : q_i(t) = c_v u(t)$
- Output pipe $r_3 : q_o(t) = k\sqrt{h(t)}$
- Level sensor $r_4 : h_m(t) = h(t)$ in the ideal case.
- Controller r_5 :

$$u(t) = \begin{cases} 1 & \text{if } h_m(t) \leq h_0 - r, \\ 0 & \text{if } h_m(t) \geq h_0 + r, \end{cases}$$

We can add an additional relation by the fact that there is a derivative of the level $h(t)$:

Derivative $r_6 : \dot{h} = \frac{dh(t)}{dt}$

The bi-partite graph of the single tank system is represented in the Figure 2.5.

Another representation in a common bi-partite graph is indicated in Figure 2.6.

2.2.2 The Structural Matrix

Applying the generic graph theory, the incidence matrix is generated from a bi-partite graph following a well-known procedure [Blanke et al., 2006]. Note that the bi-partite graph is unoriented, and therefore the following definition can be applied:

Definition 2 *The **incidence matrix** (or unoriented incidence matrix) of G is a $n \times m$ matrix (g_{ij}), where n and m are the numbers of vertex and edges respectively, such that $g_{ij} = 1$ if the vertex v_i and edge x_j are incident and 0 otherwise.*

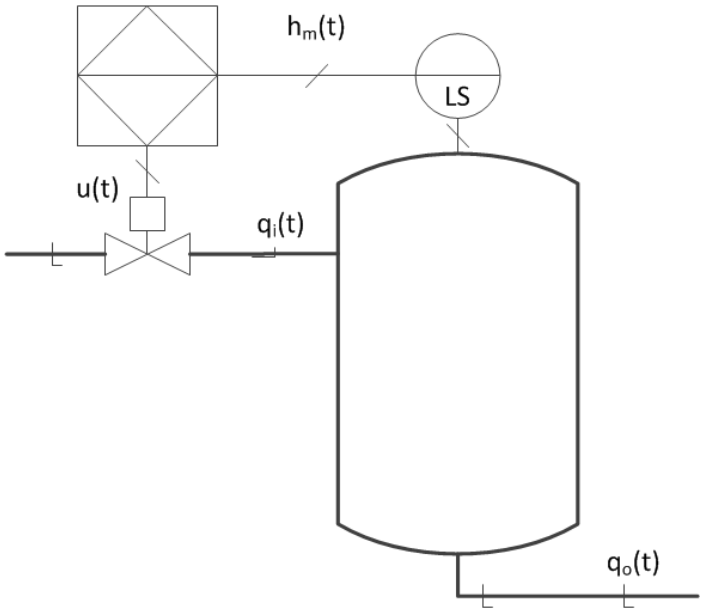


FIGURE 2.4: Single process tank example

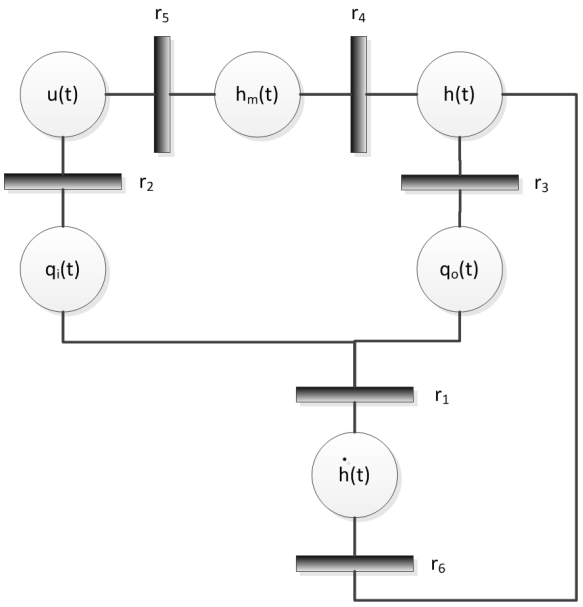


FIGURE 2.5: Bi-partite graph of the single tank system example

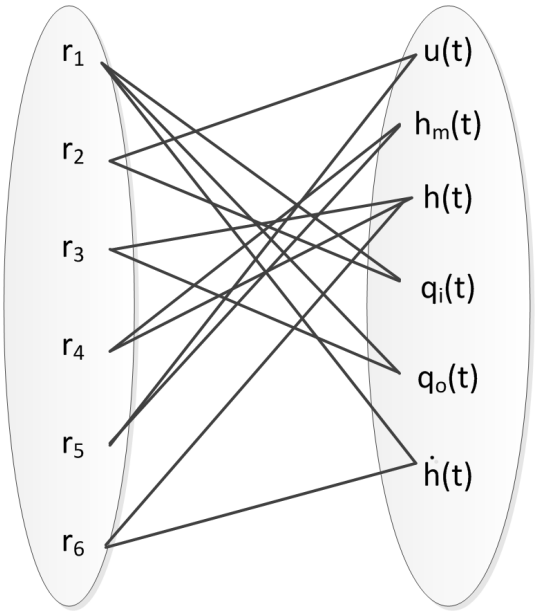


FIGURE 2.6: Common representation of a bi-partite graph of the single tank system example

For the example system of the single tank, the incidence matrix is provided in Table 2.1. It is simple to recognize that the matrix is formed by placing a "1" where there is an edge in the bi-partite graph between a process variable and a relation and a 0 when the process variable has not an edge with this relation.

TABLE 2.1: Incidence matrix of the single tank example

Relations	u	q_i	q_o	h_m	h	$\dot{h}$
$\langle r_1 \rangle$	0	1	1	0	0	1
$\langle r_2 \rangle$	1	1	0	0	0	0
$\langle r_3 \rangle$	0	0	1	0	1	0
$\langle r_4 \rangle$	0	0	0	1	1	0
$\langle r_5 \rangle$	1	0	0	1	0	0
$\langle r_6 \rangle$	0	0	0	0	1	1

It is also important to note that in a system the set process variables $\mathcal{V}_l$ can be split into known and unknown variables. Known variables will be the variables resulting from a sensor measurement or values which are already known by the controller, such as a control algorithm output. On the other hand, unknown variables will be process variables which are not measured or parameters which are not known. If the set of measured variables is identified as $\mathcal{V}_m$ and the set of unknown variables as $\mathcal{V}_u$:

$$\mathcal{V}_l = \mathcal{V}_m \cup \mathcal{V}_u \tag{2.4}$$

For the single tank example the known or measured variables are $\mathcal{V}_m = \{u, h_m\}$ and the unknown are $\mathcal{V}_u = \{q_i, q_o, \dot{h}\}$.

It will be of interest to group the variables in the incidence matrix in a way that in one side are the known variables and in the other the unknown variables. As explained in the following Section, it is a previous step in order to identify the relations which can be used to calculate the unknown variables. This incidence matrix is known in FDI as the *Structural Matrix*.

Definition 3 *The* Structural Matrix *or $\mathcal{SM}$ is defined as a matrix which rows correspond to the model relations and the columns to the process variables, sorted by known and unknown variables. Each of the matrix elements m_{ij} is set to "1" if and only if the process variable of that column j contained in the relation of the row i and is set to "0" otherwise.*

For the single tank example, the *Structural Matrix* is presented in Table 2.2.

TABLE 2.2: Structural Matrix of the single tank example

Relations	u	h_m	q_i	q_o	h	$\dot{h}$
$\langle r_1 \rangle$	0	0	1	1	0	1
$\langle r_2 \rangle$	1	0	1	0	0	0
$\langle r_3 \rangle$	0	0	0	1	1	0
$\langle r_4 \rangle$	0	1	0	0	1	0
$\langle r_5 \rangle$	1	1	0	0	0	0
$\langle r_6 \rangle$	0	0	0	0	1	1

2.2.3 The Perfect Matching

From the Structural Matrix it is possible to determine the relations which can be used to calculate the unknown variables and which relations are not essential for that. This process needs to take into account the causality of the relations with respect to the process variables contained. Therefore, considering the causality, the bi-partite graph will be oriented, that is, some process variables will participate in a relation but a subset of them will be also calculable from this relation. That is, the relation r_i is invertible with respect to the contained variable v_j.

This process of assigning unknown variables to relations which can be used to calculate it is called *matching*.

Definition 4 *A process model $\mathcal{M}$ is composed of the set of process variables $\mathcal{V}$ and the set of relations $\mathcal{R}$, and where the process variables are split into the set of known variables $\mathcal{V}_m$ and unknown variables $\mathcal{V}_u$, with $\mathcal{V} = \mathcal{V}_m \cup \mathcal{V}_u$, and being $\mathcal{A}$ the set of edges relating the process variables to the relations in the oriented bi-partite graph. The set of edges $\mathcal{A}_{pm}$ will be the subset of $\mathcal{A}$ ($\mathcal{A}_{pm} \subset \mathcal{A}$) which relate uniquely an unknown variable with a relation. The* Perfect Matching *is reached when the number of edges in $\mathcal{A}_{pm}$ is the same than the number of unknown variables, $\#\mathcal{A}_{pm} = \#\mathcal{V}_u$.*

Note that the *Perfect Matching* is not unique and several combinations are possible. Later on, it will be illustrated how the selection of the relations which are part of the Perfect Matching will determine the fault diagnosis possibilities. Note also that there are combinations which break the causality property. If instead of selecting the variable $\dot{h}$ in the perfect matching from Table 2.3, we select the process variable h. This will

TABLE 2.3: Perfect Matching of the single tank example

Relations	u	h_m	q_i	q_o	h	$\dot{h}$
$\langle r_1 \rangle$	0	0	1	1	0	1
$\langle r_2 \rangle$	1	0	①	0	0	0
$\langle r_3 \rangle$	0	0	0	①	1	0
$\langle r_4 \rangle$	0	1	0	0	①	0
$\langle r_5 \rangle$	1	1	0	0	0	0
$\langle r_6 \rangle$	0	0	0	0	1	①

imply that the value of h should be obtained from $\dot{h}$ which is not possible in all cases by solving the integration rule:

$$x_1(t) = x_1(0) + \int x_2(\sigma) d\sigma \tag{2.5}$$

where $x_1(t)$ cannot be determined if $x_1(0)$ is unknown. That is, the initial value $x_1(0)$ is not known in the general case if a real diagnostic is to be implemented and cannot be obtained as a measured variable.

An important deduction from the *Perfect Matching* is that the relations not participating can be identified as *Redundant Relations* while the set of relations participating in the *Perfect Matching* are known as *Elementary Relations*. The existence of Redundant Relations means that the system is over-constrained, that is, there are more relations than unknown variables and therefore there is analytical redundancy in the system. Up to this point, the relations could be of any type, although the causality property in most of the cases implies that quantitative relations are involved. However, the key of the diagnostic process will be to use a model with analytical relations which can be used to really compute the unknown variables and use the rest of the relations to generate the *Analytical Redundant Relations* by substituting in these relations the unknown variables by the relations which are able to compute it from known variables. These *Analytical Redundant Relations* will be used in the diagnostic of faults by generating residual expressions which evaluate 0 if the system is working nominally and will evaluate different from 0 if there is any fault.

In the single-tank example, if we observe the Table 2.3, it is possible to identify two relations which are not participating in the *Perfect Matching*: r_1 and r_5. It is easily deduced that r_5 is not providing relevant information, as in the relation only known variables are participating. However, from r_1 it is possible to substitute all unknown variables by the corresponding elementary relations from the *Perfect Matching*:

- $q_i : r_2 \rightarrow q_i(t) = c_v \cdot u(t)$
- $q_o : r_3 \rightarrow q_o(t) = k\sqrt{h(t)}$, $h : r_4 \rightarrow h = h_m$
- $\dot{h} : r_6 \rightarrow \dot{h} = \frac{dh(t)}{dt}$, $h : r_4 \rightarrow h = h_m$

which can be represented as:

$$r_1(r_6(r_4(h_m)), r_2(u), r_3(r_4(h_m))) \tag{2.6}$$

Therefore, by substituting in r_1 leads to:

$$\frac{dh}{dt} = c_v \cdot u(t) - k\sqrt{h(t)} \tag{2.7}$$

It is easy to deduce that by moving all terms to the right side of the equality, we have an equation which holds 0 if the relations evaluate the nominal behavior of the process and non-zero if there is any anomaly which is introducing a perturbation in the process. This equation is known as a residual z:

$$z \rightarrow c_v \cdot u(t) - k\sqrt{h_m(t)} - \frac{dh_m}{dt} = 0 \tag{2.8}$$

Residuals are evaluated against a threshold to determine if the deviation is because of a fault. The model uncertainties, noise and perturbations will cause the residuals to never evaluate exactly 0. Several techniques have been proposed to provide adaptive methods to calculate the thresholds as it is desired that thresholds are defined in a generic way and not depending on each implementation.

Whatever technique is used to evaluate the residual, eventually in the ideal case in a process with no faults the complete set of residuals will be evaluated as 0 while in a presence of a fault some residuals will evaluate different from 0. These residuals are compared with a threshold providing the result value 1 if the threshold is passed and 0 otherwise. These results grouped in a vector define the *fault vector*. Formally if $\tau(z_i)$ is the threshold evaluation function of the residual z_i, the fault vector will be the vector $f = \{\tau z_1, \cdots, \tau z_n\}$.

This vector is the source for the fault diagnosis using the structural analysis. The vector will be evaluated in real-time and a component in the vector different from "0" will indicate the presence of a fault in the system.

2.2.4 The Fault Signature Matrix

In the previous Section, we have defined how to obtain a set of residuals from the *Structural Matrix* by following the process to determine the *Perfect Matching*. As a result of this process, a set of relations are identified as redundant relations and another set as elementary relations. The set of redundant relations can be transformed in a set of residuals which evaluate 0 if there are no faults present in the system. On the other hand, it is important to remark that the relations can be always associated to one or more components in the system. Therefore, the set $\mathcal{C}$ of components is introduced in the model and it holds that a bi-partite graph can be also obtained between the components and the relations. This is trivial, since the relations have been obtained from the process model and specifically from the components forming the process.

Definition 5 *The* Fault Signature Matrix *or $\mathcal{F}$ is a matrix where the rows are the residuals $\mathcal{Z}$=$\{z_1, \cdots, z_n\}$ and the columns are the process components $\mathcal{C}$=$\{c_1, \cdots, c_n\}$, where we set the value "1" to the position f_{ij} if the component c_j is sensitive to the residual z_i and "0" otherwise.*

Note that the sensitivity is determined in a way that if any change in a process variable of any of the relations belonging to a component can affect the calculation of the residual, we conclude that a fault in this component will be detected with this residual. In this sense, in the single tank example, the controller output $u(t)$ is also influencing the calculation of the residual r_1 and therefore a fault on the controller will also imply a perturbation which will result in evaluating the residual as non-zero.

The *Fault Signature Matrix* of the single-tank example is the most simple, as it has only one row and five columns corresponding to the components set C = {tank, valve, output pipe, level sensor, controller} and it has the value "1" in all columns, as all relations are used to calculate the residual.

From this definition of the *Fault Signature Matrix*, it can be deduced that in the Structural Analysis, faults correspond to deviations in the evaluation of component relations. Therefore, the only detectable faults will be faults which indeed cause a deviation in the evaluation of any of the component relations. It is possible to identify two properties from the *Fault Signature Matrix*:

- A relation deviation is structurally detectable if and only if it causes a non-zero value in some residual z.
- A relation deviation is structurally isolable if and only if it has a unique signature in the *Fault Signature Matrix*.

The first property provides the capability of identifying the occurrence of a fault but does not allow the identification of the faulty component, while the second, if in the *Fault Signature Matrix*, it is possible to define a different signature for each component and it will be also possible to identify the faulty component. Obviously this is not always possible and in most of the systems there will be common signatures for several components. For the simple case of the single-tank system, the first property is hold as any deviation in a component relation will cause to evaluate the residual non-zero, however it will not be possible to identify which is the faulty component.

The example of fault signature matrix in Table 2.4 relates the set of components $\mathcal{C} = \{c_1, \cdots, c_6\}$ with the set of residuals $\mathcal{Z} = \{z_1, \cdots, z_5\}$. It can be verified in the Table that each component has a different fault signature, i.e. the Hamming distance between any combination of two component fault signatures is bigger than 0. Formally, being $f(c_i)$ the fault vector of the component c_i and $\oplus$ the Hamming distance:

$$\forall c_i \in \mathcal{C}, \nexists c_j \in \mathcal{C} | f(c_i) \oplus f(c_j) = 0 \tag{2.9}$$

TABLE 2.4: Example of a fault signature matrix

Residuals	c_1	c_2	c_3	c_4	c_5	c_6
$\langle z_1 \rangle$	0	0	1	1	0	1
$\langle z_2 \rangle$	1	0	1	0	0	0
$\langle z_3 \rangle$	0	0	0	1	1	0
$\langle z_4 \rangle$	0	1	0	0	1	0
$\langle z_5 \rangle$	1	1	0	0	0	0

2.2.5 Evaluating the Fault Vector

There are at least two different ways of evaluating the fault vector f to implement the fault diagnosis. The simplest way is to calculate the Hamming distance of the vector with the fault vectors of each component and consider as the faulty component the component with the minor distance to the fault vector. However, it is clear that this method provides the same importance to the consistent residuals than to the inconsistent residuals. A consistent residual, that is, a residual which evaluates as 0, is not conclusive as the sensitivity of the residual to a fault depends on the fault and the fault magnitude [Blesa et al., 2014]. Therefore, there is a better approach which is to compare only the inconsistent residuals row by row and give a positive score point if the value matches and a negative score point if the value is different, and finally sum all points for each component column. The estimated faulty component will be the one with more score points.

Considering a process with the *Fault Signature Matrix* in the Table 2.5, and the fault vector $f = \{1, 1, 0, 0\}^T$, the Hamming distance of the fault vector to each of the columns is $d = \{1, 2, 1\}$ while if we apply the second method we obtain the following result $s = \{0, 0, 2\}$. In the first case, the component c_1 and c_3 have the same Hamming distance and it is not possible to distinguish between both components. However, in the second case as the inconsistent residuals are most significant, the result is that the fault estimation indicates that is the component c_3 the faulty component.

TABLE 2.5: Estimation of the faulty component example

Residuals	c_1	c_2	c_3
$\langle z_1 \rangle$	1	0	1
$\langle z_2 \rangle$	0	1	1
$\langle z_3 \rangle$	0	0	1
$\langle z_4 \rangle$	0	1	0

2.3 Agent Based Systems

Agent based systems will be systems which are supervised, managed or controlled by software agents or with the participation of software agents. The design of multi-agent systems is a relatively new approach of the Software Engineering which allows the software components not only to encapsulate the methods and data (Object Oriented Systems) but also the behavior [Wooldridge, 2002]. Nowadays software agents are also known as Active Components. This Section revises the current architectures of the software agents in order to select the best option to implement FDI supervision using software agents. In this way, the review goes into the implementation details of the selected architecture in order to know the available functionalities and how can be used.

2.3.1 Software Agents Characteristics

The main characteristics of the software agents are:

- Autonomy: An agent is autonomous in the sense that it does not need an external intervention (in particular a human intervention) to react upon an event, to control its own actions or to control its own state.
- Reactivity: An agent receives information (signals) from its environment and reacts according to this information.
- Proactivity: A agent not only will start actions as a response to changes in its environment but also will exhibit a behavior oriented to objectives even taking the initiative.
- Sociability: An agent might require information or services from other agents (systems or humans) to accomplish its tasks and reach its objectives.

An important characteristic of the agents is that they do not need to know all possible situations (or states) to define an action plan. An agent can be considered an stochastic state machine, since the following state cannot be determined a priori. The following state will depend on the current state and the stimuli received by the agent in this state. In the case of a deliberative agent, before moving to the following state there will be also a deliberative phase to determine the following action.

A traditional software component receives external stimuli from the environment in form of events. Each event will contain associated information, which can be very simple, as the activation of a switch, or very complex, as for instance the reception of a data frame. The software component reacts executing in a reactive form the algorithm which has been pre-programmed in the design phase and will provide a response in function of that. The system is totally deterministic and for the same internal state, given the same inputs it will provide the same outputs.

The importance of software agents remains in that the agent has an internal representation of the external world and will react to changes to this world in a spontaneous way. This representation will be implemented in the Knowledge Base or Beliefs Base. On the other hand, the agent programming is implemented based in objectives. To determine the agent structure first it is required to define the objectives or goals which have to be reached to solve the laid problem. Once the objectives are defined, the mechanisms to reach these objectives will need to be identified, these mechanisms are normally known as plans. An agent can have multiple plans to reach an objective. The selection of the most suitable plan to be applied to reach the objective is a deliberative function which can be implemented in several ways, for instance, using optimization functions.

2.3.2 Software Agent Architectures

The different architectures to implement software agents which have been proposed depend on the ability to respond to decisions based on a natural reasoning, somehow reproducing the human reasoning. This fact facilitates the modeling and the capacity of abstraction. The software agent architectures can be split into two main groups: the architectures of deliberative agents and the architectures of reactive agents. The deliberative agents have a symbolic representation of the real world and decide their actions (plans) based on the information they received, the knowledge base and a set of rules. The reactive agents, on the other hand, do not contain this internal representation of the real world nor their actions are based on the past history nor plan future

actions based on the current status, they simply respond to the stimuli received in real-time evaluating the results and using mechanisms such as the *reinforcement learning* to deduce which actions have been more successful.

Nowadays the most popular way of implementing agents is an intermediate approach, that is, mixing the deliberative and reactive architectures. This approach allows the agent to react quickly or in a more deliberative way depending on the event received and the timing required to provide a response.

BDI Architecture

The most popular agent architecture to implement multi-agent systems is based on the human reasoning theory proposed by Bratman [Bratman, 1999]. BDI stands for Believes, Desires and Intentions, which match with the definitions of Knowledge, Objectives and Plans. This architecture is based on defining the mental attitudes of an agent in these three concepts.

- The Knowledge (*Believes*) corresponds to the knowledge that the agent has of the real world in the current instant. It is normally represented as a database which is updated in real-time from the events (or *stimuli*) that the agent receives. It is also known as Believe Base and it is usually implemented as a set of structured data or even as a relational data-base. It can also include a set of rules which can be executed periodically or by changes in the data caused by internal or external effects.
- The Objectives (*Desires*) define the final purpose of the actions undertaken by the agent.
- The Plans (*Intentions*) are the representation of the set of actions which an agent can execute to accomplish its objectives.

AOP Architecture

This architecture was proposed by Shoham in [Shoham, 1993] and is based on defining the mental state of an agent through an specific programming language named Agent-0. This language allows the definition of the believes (the knowledge), the capacities and a set of inference rules which will define the behavior of the agent in the form of commitments. Initially the Agent-0 language had several drawbacks, as for instance that the complete specification should be implemented in LISP, it did not have available planning utilities, the agent actions were executed sequentially and it was not possible to implement the interaction with remote agents. However, recently some extensions and improvements have been implemented which allow these functionalities and solve some previous problems [Eduard Muntaner, Acebo, and Rosa, 2005].

SOAR Architecture

These architecture allows the creation of deliberative agents with its own symbolic representation of the real world. The architecture define a set of components which form the deliberative capacity of the agent:

- States: Maintain the information of the current situation.
- Operators: Set of operations which are the means to progress through the problem space.

- Working Memory: Set of data which contain the hierarchy of states and the operators which have been applied up to now.
- Long-term Memory: Set of data which form the repository used by the architecture to generate the behavior.
- Motor Interface: Component which translate the perceptions of the real-world to the corresponding symbols in the internal representation.
- Decision Cycle: Process which continuously selects the more appropriate operator to apply to the current state to pass the the next state according to a global objective.
- Impasses: Situations in which more knowledge is required to resolve the situation and progress towards the target objective.
- Knowledge mechanisms: Set of mechanisms as Chunking, Reinforcement Learning, Episodic Learning and Semantic Learning [Lehman, 1996].

The development structure of this type of agents is similar to the development environments of some expert systems such as CLIPS. An updated implementation is described in [Laird and Congdon, 2006].

2.3.3 Multi-agent System Platforms

In this Section, some software agents development platforms will be presented. The selection of a platform to implement software agents is a critical decision. The development environment needs to be mature enough to allow the developer to focus on the implementation of the agents and not having to devote a big effort just to use the tools. Also it needs to be supported in some way, allowing the request of support in case it is needed. If the development platform has currently some implementations working in the real-world it is also a positive fact. Note that once a platform is selected, there will be a lot of effort invested which will depend on the proper functioning of this platform.

JACK

JACK is a complete development environment to implement agent-based systems. It is based on the BDI architecture. Define and extension of the Java language to program the agent structure, the agent plans, the events and the data structures. The language is pre-processed to generate the class structures in the Java language which finally will be executed in the corresponding Java Virtual Machine. JACK is a commercial product for which the source code is not available [*JACK Documentation*].

JADE

JADE is a set of libraries and utilities which facilitate the development of multi-agent systems, including an execution platform (runtime). The implementation is programmed in the Java language which allows to deploy multi-agent systems on several hardware platforms were there is the availability of a Java Virtual Machine. The tool does not anticipate any specific hardware platform. The exchange of messages between agents is implemented through ACL, a specification which has been standardized by the FIPA organization [Caire, 2007].

JADEX

JADEX allows the implementation of agents following the BDI architecture. It defines the data structures to specify the knowledge (believes), the desires (objectives) and the intentions (plans). The platform can use JADE and allows also to be adapted to other execution environments (runtime). The definition of the agents is implemented in XML and the implementation in Java. JADEX incorporates a model of behavior of the agents, reacting to changes in the knowledge base, the introduction of new plans or incorporate changes in the target objectives. The platform also includes a deliberative model to select the plan to execute at each moment in function of the current objectives [Braubach, Lamersdorf, and Pokahr, 2003].

RETSINA

This platform also follows the BDI architecture model, implementing a set of classes which support the generation of the behavior of the agents and the communications between them. A difference with the other platforms is that the development environment is Microsoft Visual C and the classes then are developed in C++. It does not include a reasoning engine and the communications language is based in KQML. The agents can be executed on the operative systems Windows, Solaris and Linux [Michael Rectenwald, 2002]

ARTIS/SIMBA

This platform has a specific characteristic which is that it takes into account the response time of the agents. In order to implement this characteristic, it is needed that the agent is executed in a real-time platform and therefore that takes into account the time restrictions when executing the agent functions. The research group GTI-IA of the *Universitat Politècnica de València* (UPV) has developed a platform of multi-agent systems in real-time named SIMBA [Julian et al., 2002]. The agents of the SIMBA platform are developed with the ARTIS platform. This architecture include extensions to work in systems with strong restrictions in real-time. The implementation language is C++ and the execution platform is RT-Linux.

3-APL

The triple-APL (Artificial Autonomous Agents Programming Language) defines a programming language and a set of tools to define multi-agent systems based on the BDI architecture. As a difference with Jack and others, it allows the definition of the desires, capacity, knowledge, plans, objectives and rules with the high-level language PROLOG in an easy way. The agents are implemented in Java and the platform incorporates an interpreter for the BDI specifications. The environment can be executed in several operating systems as Windows, Linux and Solaris. The agents can be distributed and communicate between them through the network from the different platforms. This platform includes a component where agents are registered called AMS (Agent Management System). It also allows the interaction with agents developed with other architectures since the communications are implemented following the standard FIPA [Dastani, 2006].

2.3.4 Architecture of the BDI Agents

In previous Section, it has been explained that the agents are software components which allow the systematic development of intelligent behavior. The objective of agent based programming is to provide an structure which allows the implementation of algorithms of decision-making in a natural form and easy to understand. The complexity of an agent should come from:

- The complexity of the problem to be solved by the agent.
- The complexity of the platform used to create the agent.

In this work, the agents are introduced as a natural solution to the distributed diagnosis problem. In this Section, the components and functions of an agent are presented and it will be demonstrated how they fit simply and in a comprehensible way with the requirements to implement a systematic fault diagnostic system.

The BDI architecture has been finally selected. The main reasons are: its extensibility, its wide application and its maturity.

In general, the problems of the BDI architecture are:

- The lack of mechanisms to implement machine-learning.
- It does not incorporate explicit mechanisms to interact with other agents.
- The architecture does not incorporate the forward-planning and it is not possible to evaluate a priori the results of the execution of a plan.

The reasoning basic algorithm of the BDI agents it is known as the Procedural Reasoning System (PRS) and is indicated below:

1. initialize-state
2. repeat
   (a) options: option-generator(event-queue)
   (b) selected-options: deliberate(options)
   (c) update intentions(selected-options)
   (d) execute()
   (e) get-new-external-events()
   (f) drop-unsuccessful-attitudes()
   (g) drop-impossible-attitudes()
3. end repeat

The agent architecture can be represented as a set of states and the allowed transitions between these states. The architecture presented is based on the extended BDI architecture presented by Pockahr in [Pokahr, Braubach, and Lamersdorf, 2005]. This architecture breaks-down the generic PRS algorithm in sub-parts, which are only invoked when required. These sub-parts are known as meta-actions.

The platform selected to implement the Supervisor Agent is JADEX. This platform has reached an important relevance either in the academic world as with implementations which solve real problems [Braubach and Pokahr, 2007]. Furthermore, it is an open-source project and the code of the whole platform is available.

In JADEX, the agent architecture will define in which form it will be decided which is the next action that an agent has to execute and when new actions shall be initiated.

The interpreter which will perform these actions is based on the following structures:

Agenda

The Agenda are the data structures which contain the actions which have to be executed. The interpreter will select the next action to execute and will execute it, changing the internal state of the agent. The execution of an action may lead to the creation of new actions which will be inserted into the agenda or may affect the order in which the previous actions were in the agenda, put on hold some actions or even drop actions from the agenda which have become obsolete. Thus, the objectives could have changed or the context of execution for an action could have changed.

The agent state $\sigma \in \Sigma$ defined by the tuple $\langle B, \Gamma, \Pi, A \rangle$, where:

- B is the knowledge base.
- Γ is the set of agent objectives.
- Π is the set of agent plans.
- A is the set of planned actions in the agenda $\{\alpha_1, \alpha_2, \cdots\}$.

An action will be defined by the tuple $\langle \tau, \phi_1, \phi_2, \cdots \rangle$, where τ is the type of actions and $\phi_1, \phi_2, \cdots$ are the action parameters. The name and type of parameters are determined by the type of action.

For each type of action, the following characteristic functions are introduced:

- p_{pre} is the pre-condition function which determines if the action is still valid in the current context.
- f_B is the transition function which describes the changes in B (the knowledge base).
- f_Γ is the transition function which describes the changes in Γ (the set of objectives).
- f_Π is the transition function which describes the changes in Π (the set of plans).
- f_A is the transition function which describes the changes in the set of actions planned in the agenda.
- f_{eff} is the effect function which determines the set of next actions which should be added to the agenda.

A state transition caused by the execution of an agent action $\sigma \in \Sigma$ according to these function definitions can be represented as: $\sigma \xrightarrow{\alpha} \sigma'$, with $\sigma = \langle B, \Gamma, \Pi, A \rangle$, $\alpha \in A$ and $\sigma' = \langle B', \Gamma', \Pi', A' \rangle$. And therefore:

$$B' = f_B(\alpha, \sigma)$$
$$\Gamma' = f_\Gamma(\alpha, \sigma)$$
$$\Pi' = f_\Pi(\alpha, \sigma)$$

$$A' = A \backslash \{\alpha\} \cup f_{eff}(\alpha, \sigma)$$

Other external sources can add actions $\{\alpha\}$ to the agenda, as for instance a message received from another agent. These new agenda entries will not affect any other component of the agent, therefore, the state transition in this case will be defined by $A' = A \cup \{\alpha'\}$.

In order to define the process to select the next action to execute from the agenda, the function $f_{sel}(A)$ is implemented. This function can be as simple as for instance take the next action from the list, but it is possible also to complicate it by adding the possibility of introducing priorities. This capability is important as some mechanism is required to schedule the execution of actions in a real-time environment, that is, with time constraints.

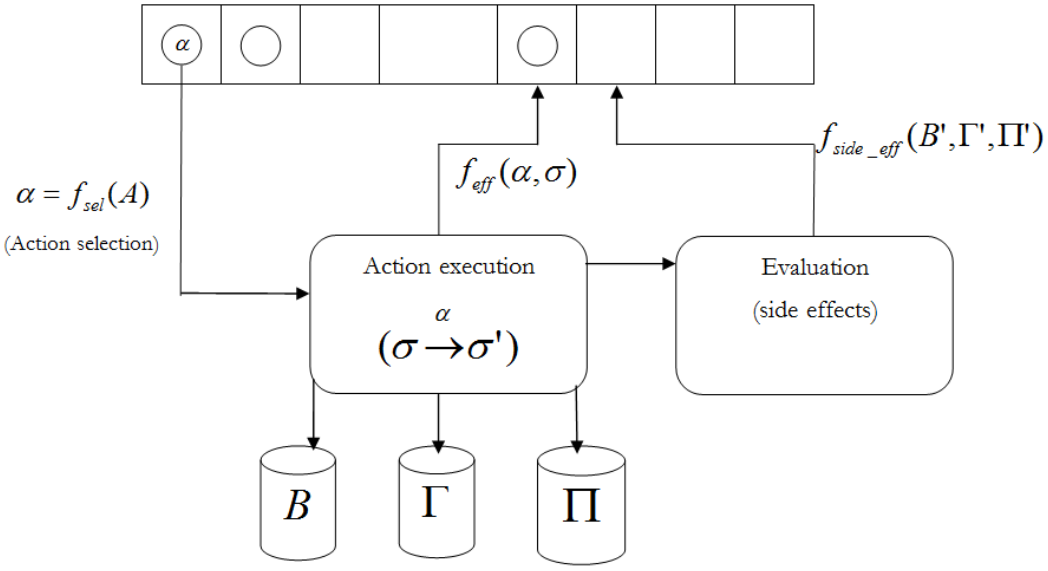


FIGURE 2.7: BDI agent actions sequencer

Events

In order to understand the planning mechanism of the actions in the agenda, it is important to note that this architecture is driven by events ε. There are two types of events:

- External events: correspond to those events generated by the reception of a message from another agent.
- Internal events: correspond to those events which are generated as the result of the execution of a plan, by changes in the knowledge base or by changes in the objectives of an agent.

The events can be handled by plans already instantiated and running or trigger the creation of new plan instances from a template, according with the intentions of each plan. That is, each plan declares the intention of handling one or more events, in a way that when an event is received the plan requests its activation to handle it. It can happen that there are no plans with a declared intention to handle an events and in this case the event is simply ignored.

An agent will implement an events queue which will be handled and by default this queue is implemented following the FIFO algorithm (*first-in first-out*).

The Plan Set

The plan set of an agent Π is formed by instances of plans and plan templates. The plan instances will have the form $\pi = \langle pt, \varepsilon, \mu \rangle$, where pt is the plan template, ε is the current event being handled by the plan and μ is a counter which indicates in which is the current plan execution step. The plan templates contain two different parts: The header and the body. The header contains the parameters which indicate in which conditions the plan should be activated (the set of intended events) and the plan priority. The priority will determine which plan will be selected for execution, in case that there are several plans which intend to handle the same event or have the same execution conditions. The body of the plan contains the plan instructions grouped in steps.

The Meta-actions Definition

In order to select the plans which will be executed, the following actions are defined:

- Find applicable candidates α_{fac}: Obtains from the plan set all plans Π_{app} which are candidate to handle an event ε arrived to the system and which do not have any event to handle yet i.e. $\langle pt, \perp, \mu \rangle$, where μ is the step counter.
- Select candidates α_{sc}: This function will obtain from the plan set (instances or templates) the plans Π_{can} which are candidate for execution. For instance, evaluating the plan execution pre-conditions. For the selected plan templates pt which become candidates for execution a new instance will be generated $\pi = \langle pt, \perp, 0 \rangle$ indicating that an event is not yet assigned ($\perp$) and with the step counter 0.
- Schedule candidates α_{schc}: This action updates all selected plan instances $\pi \in \Pi_{can}$ to include the event to be handled, thereby adding newly created plan instances to the plan set. In addition a new α_{eps} execute plan step action is added to the agenda for each of the selected plan instances.
- Execute plan step α_{eps}: This action will execute a step of the plan. This function can change any aspect of the agent. Thefeore, it can contain any of the previously indicated functions which modify the agent state: f_{B_π}, f_{Γ_π}, f_{Π_π}. In addition, the agent step counter will be incremented in 1 and a new action α_{eps} will be added to the agenda. In case that it is the last step of the plan, it can happen than the instance is deleted or keeps waiting another event. In any case, with the last step no further actions are introduced in the agenda.

2.3.5 Agent Goals

An important point of the BDI architecture is that the agent goals are defined explicitly. Designing an agent based on the objectives which should reach is a very simple and natural task. For instance a garbage collector robot the list of goals would be: {patrol, charge_batteries, collect_garbage}. In order to define which goals should pursue the agent at each instant, it is possible to define pre-conditions which will indicate the validity of the goal in the current context. In the garbage collector robot, it could be defined that it should "collect garbage" during the day and "patrol" during the night. In this way, the context of the goal "patrol" would be only valid in the night hours

and the context of the goal "collect garbage" will be only valid during the day hours. In addition, the architecture allows the definition of relations between goals which at design level delimit the goals which can be pursued at the same time. For instance, in the example, the goals "patrol" and "collect garbage" are opposite to "charge batteries". That is, if the context of the goal "charge batteries" is valid, the activation of this goal will override the goals "patrol" and "collect garbage".

The goals can be created and dropped at execution time. A condition defined over the knowledge base, for example, can generate the action to create a new goal. This new goal will pass to a pre-defined state depending on its validity pre-condition. In this way, the agent goals, once created, will follow a life-cycle based on three states: option, suspended and active. See Figure 2.8.

- Option: goals in the option state are the goals which momentarily cannot be pursued, for instance, because they are in conflict with another goal.
- Suspended: Goals are suspended because they cannot be executed in the current context. That is, the execution pre-condition does not hold. When the context will change making the execution context pre-condition valid, the state will be updated to option.
- Active: Active goals are the goals which were in the state option and which can be pursued at this moment and in this context because are not in conflict with any other goal and their context is valid.

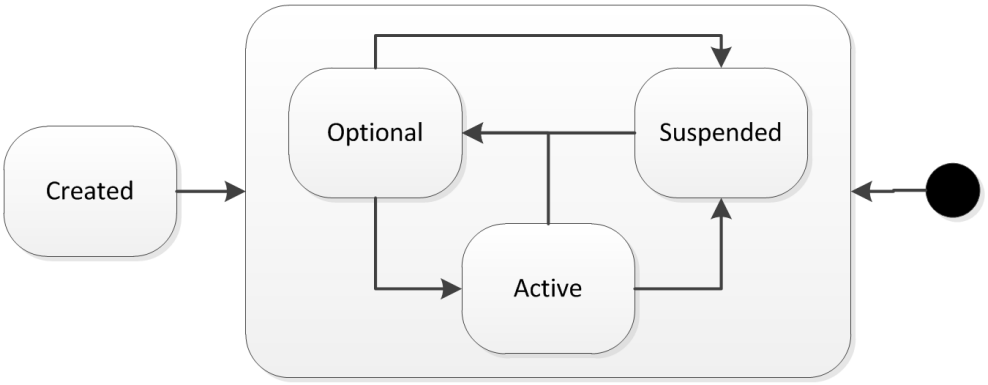


FIGURE 2.8: BDI agent objectives life-cycle

In order to manage the goals creation, the new type of action *CreateGoal* is added, $\alpha_{cg} = \langle CreateGoal, gt \rangle$, where gt is the objective template (*goal template*). A new goal $\gamma \in \Gamma$ will be composed of:

$$\gamma = \langle gt, s \rangle, s \in \{optional, active, suspended\}$$

That is each goal instance will originate from a specific goal template and will be in a specific state.

This action α_{cg} will be always applicable and will only affect the goals set, adding a new one, leaving the rest of agent components unchanged.

$$f_\Gamma(\alpha_{cg}, \sigma) = \Gamma \cup \{\gamma\}, \gamma = f_{create}(gt)$$

When a context change is produced which affects a goal, the goal context has to be evaluated to determine if it is still valid or the goal state has to change to suspended or option. In order to implement these goal state changes, a new action *SwitchContext* is added, $\alpha_{sc} = \langle SwitchContext, \gamma, s' \rangle$, where s' is the new state to be achieved, $s' = \{suspended, option\}$. This action only has the pre-condition that the goal is not already in one of the target states. This action will only change the goal set as follows:

$$f_\Gamma(\alpha_{sc}, \sigma) = \Gamma \backslash \{\gamma\} \cup \{\gamma'\}, \gamma' = \langle gt, s' \rangle$$

Finally a new action is added to drop a goal from the goal set $\alpha_{dg} = \langle DropGoal, \gamma \rangle$ leaving the goal set as follows:

$$f_{Gamma}(\alpha_{dg}, \sigma) = \Gamma \backslash \{\gamma\}$$

Goals Deliberation

In order that an agent with multiple goals which should be pursued at the same time do not conflict among them, the agent must follow a deliberation process. A simple deliberation process is presented which is known as *Easy Deliberation* and is based in two type of properties: the cardinality and the inhibition arcs.

- Cardinalities: Restrict the maximum number of goals of the same type which can be active at the same time.
- Inhibition arcs: Define the relations between goals which imply a negative effect with respect their activation. That is, if an goal is active and another goal needs to pass from option to active but it has an inhibition arc with the one which is active, this second goal cannot be activated.

There are two situations in which the deliberation algorithm must be activated: the first is when a goal is newly created and passes to the option state or when a suspended goal passes to option because its execution context becomes valid, in these cases the algorithm should decide if the new option goal can be activated and which are the consequences of activating it, that is, if any of the currently active goals need to be suspended. The other situation is when for any goal its execution context becomes not valid and must be suspended. In this case, the algorithm should decide which from the optional goals which were inhibited can be now activated. In order to manage these two situations the actions *DeliberateNewOption* α_{dno} and *DeliberateDeactivatedGoal* α_{ddg}.

The *DeliberateNewOption* is responsible for activating a goal in the option state $\gamma_o = \langle gt_o ptional \rangle \in \Gamma_o$ when its execution context becomes valid. Therefore, the function has to test the cardinality and the inhibition arcs with the currently active goals by checking the predicate $p_{act}(\gamma_o)$ defined as:

$$p_{act}(\gamma_o) : \Gamma_o \to \{true, false\}, p_{act}(\gamma_o) = \forall \gamma \in \Gamma_\alpha (\gamma \nrightarrow \gamma_o) \wedge |\Gamma_\eta| < f_{card}(gt_o)$$

with $\Gamma_\eta = \{\gamma = \langle gt, active \rangle \in \Gamma_\alpha | gt = gt_o \wedge \sigma_o \nrightarrow \sigma\}$ and $f_{card}(gt_o) : \Gamma \to \mathbb{N}$ the cardinality function and $\to \subseteq \Gamma \times \Gamma$ the inhibition relation.

$$f_{Gamma}(\alpha_{dno}) = \Gamma \setminus \{\gamma\} \cup \{\langle gt, active \rangle\} \setminus \Gamma_{inh} \cup \Gamma_o$$

The predicate $p_{act}(\gamma_o)$ is true when there is no active goal that inhibits goal γ_o, i.e. no pair $(\gamma, \gamma_o), \gamma \in \Gamma_\alpha$ is part of the inhibition relation $\rightarrow \subseteq \Gamma \times \Gamma$ and the number of goals of this type which can be active Γ_η is lower than the allowed cardinality of this goal defined by the function $f_{card}(\gamma_o)$. Note that the set Γ_η needs to remove the goals which will be inhibited when the goal γ_o becomes active. This set is defined as $\Gamma_{inh} = \{\gamma \in \Gamma_\alpha | \gamma_o \rightarrow \gamma\}$.

In addition, another meta-action *DeliberateDeactivatedGoal* α_{ddg} is required to check for any goal which has been deactivated if there are goals in the Γ_o set which could benefit from that.

$$f_{eff}(\alpha_{ddg}, \sigma) = \{\langle DeliberateNewOption, \gamma_x \rangle \, | \gamma_x \in \Gamma_{inh}\}$$

The set of inhibited goals is defined as Γ_{inh} and is calculated with the function $f_{inhibit}(\gamma, \gamma_x)$.

To illustrate this functionality, the example of the garbage collector robot can be used. In the most simple defintion, we can define 3 goals:

$$\Gamma = \{ChargeBattery, CollectGarbage, Patrol\}$$

for the agent controlling the robot. The activation context for the Collect Garbage goal is the day time range $[08:00] < t < [17:00]$ and the activation context for the Patrol goal is the night time range $[17:00] \leq t \leq [08:00]$ while the execution context for the Charge Battery goal is that the battery level is below 5%. The Charge Battery goal has an inhibition arc with each of the other two goals and the cardinality of all goals is 1, meaning that only one goal instance of each type can be active at a time. The Easy Deliberation algorithm will exactly implement the expected behavior for the agent, during the day will collect garbage until the activation context of the Charge Battery goal becomes true, then it will become the active goal deactivating by inhibition the collect garbage goal. Note that once the goal Charge Battery becomes active the execution context has to be maintained true until the battery reaches the level 100%. Then, the Charge Battery goal activation context is not valid any more and the goal becomes option again. The *DeliberateDeactivatedGoal* function will be executed selecting from the option goals which can be activated again, and depending on the current time; day or night, it will activate the Patrol or Collect Garbage goal.

Goal Types

The architecture implements four types of goals depending on the definition of the activation context:

- Perform Goal: This goal will define an action which must be executed. While the goal is active the agent will be executing this action by executing the associated plan. The goal "Patrol" is a Perform Goal.
- Achieve Goal: This goal will become active until a specific condition is reached. That is, while the condition is not reached, it will be active and all matching plans will be executed until the condition is met or error.

- Query Goal: The function of this goal is to reach some specific information and finishes when this information is provided. That is, while the information cannot be obtained it will continue executing all matching available plans until the information is reached or error.
- Maintain Goal: This goal implements the function of maintaining a defined state. This is expressed as a condition which should be maintained (for example battery level > 20%). Additionally, another condition can be defined which indicates the target condition once the condition to be maintained is violated, in the case of the garbage collector robot the target condition would be battery level = 100%. This goal will never expire unless it is explicitly eliminated.

Chapter 3

FAST Architecture

In [Blanke et al., 2006], the component-based FDI analysis and architecture is introduced. This architecture defines a generic component model based on services and use modes. The use modes define which subset of services is available at each mode. A component defines a physical entity with physical characteristics and a component model is exactly modeling those characteristics. The components can be aggregated and both, the analysis of faults and fault propagation are extended to the aggregation.

However, sometimes the concept of component aggregation is not enough. In our work, an aggregation of components will be a process, and the objectives of the process and its constraints are not component properties but process properties. These two different layers can enhance the FDI architecture since component (or component aggregation) objectives and process objectives are not always related. In a fermentation reactor, for instance, each component (tank, valve, sensor, pump, heater, heat-exchanger, etc.) has a very specific objective. However, the purpose of the process is to maintain the environmental variables at fixed set points in order to perform the fermentation in optimal conditions, hence a more complex model is required to monitor the process objective. This complex model could be determined by using well known techniques (first principles, estimation, identification, etc.), especially if the system is already existing. In this example, the model can define the BOD (Biological Oxygen Demand) in the reactor in function of the time, the environment variables (temperature, pH, pressure) and some chemical compounds concentration.

Therefore, a process is described as the aggregation of components with a global objective. From this definition, it is possible to study even the aggregation of processes, where higher level objectives are defined. The set of aggregated processes will represent a process plant involving higher level objectives and also the evaluation of the FDI process in a distributed manner.

In [Bouamama et al., 2005] and [Merzouki et al., 2013], model based FDI and FTC are introduced using bond-graphs as a helpful modeling paradigm. Moreover, some tools ([Blanke and Lorentzen, 2006], [Bouamama et al., 2005]) allow the generation of the process model by connecting blocks and they can generate *C* code for the model. These tools are mainly oriented to the modeling of the process and some of them perform the Structural Analysis. *FAST* is not a modeling tool but a tool to implement a FDI system that can be tested in simulation and deployed to an on-line implementation with a minimal effort. The only requisite is that the models of the process components must be already available in the tool. They can be expressed in any form (as e.g. expressed by means of analytical equations or bond-graphs) being easily integrated in *FAST* just by implementing the interface as specified in the Appendix B. In this case, the tool utilization is very simple: the process engineer does not need to deal with modeling, but only indicating the topology of the process (components and their relationships) as indicated in a P&ID diagram it is possible to obtain valuable design

information (by simulation) which can be feed-backward into the design to refine it. Finally, it is possible to connect the tool to the process and implement FDI in real-time without needing to add some extra modeling information. For this latter feature, the tool implements an OPC interface which allows the direct interaction with any commercial SCADA. The tag names, which establish the relation between the process data and *FAST*, are provided in the Process Definition File (PDL).

The low level FDI analysis implemented in *FAST* related to process components has been introduced in [Duatis, Angulo, and Puig, 2014], where the algorithms to perform the Structural Analysis from the process components point of view are explained and verified using the well-known two-tanks example. The main software architecture characteristics of *FAST* are the elements that will be introduced in detail in this Chapter:

- The aggregation of components with shared objectives is defined as a process.
- It is possible to define process objectives in the form of relations.
- Components are abstracted like software structures (Java classes).
- The software representation of the components are basically data structures which do not implement behavior since are only an abstraction of the real component associated to one or more relations.
- Services belonging to each component are generalized by the definition of relations. Each component has a predefined set of relations. Each type of relation is represented also as a class. Several input process variables are implied in each relation but for the process simulation only one output process variable is calculated.
- Each process variable is also represented as a class.
- The real-time FDI behavior is provided by a software BDI agent which is the responsible of the supervision of a process. Software agents are developed in the JADEX platform [Braubach and Pokahr, 2007].
- The aggregation of components is the world representation of this agent.
- Processes can be distributed in subsystems.

The *FAST* tool is implemented in the Java language and is composed of three main software components:

- The *FAST Libraries*
- The *FAST Simulator* and,
- the *FAST Supervisor Agent*.

The *FAST Simulator* is the component responsible of loading the process definition from the Process Definition File (PDL) generating the Process Model and performing the FDI Structural Analysis as defined in [Duatis, Angulo, and Puig, 2014] following the approach proposed in [Staroswiecki and Declerck, 1989], [Blanke et al., 2006], and [Isermann, 1997]. In this way, the *FAST Simulator* is able from the simple definition of a process to obtain the Structural Matrix $\mathbf{M}$ and the Fault Signature Matrix $\mathbf{F}$. Although other tools have been proposed in [Bouamama et al., 2005] and [Blanke and Lorentzen, 2006], the *FAST Simulator* is not a modeling tool, but a tool to carry out the design of

a FDI system for a process and to evaluate its diagnosibility, allowing eventually to perform a final FDI implementation by connecting *FAST* to the process by means of the OPC interface.

In this Section, the architecture of the *FAST* tool will be presented based on the first ideas indicated in [Duatis, Angulo, and Puig, 2015]. First the components will be described from a functional perspective, being the most important; the Process Definition File and the Process Model. The PDL structure will be defined in Section 3.1. This file provides the process information required to build the process model based on components, relations and process variables. Then, the Section 3.2 will present how the process model is generated by the software. The process model is a component part of the *FAST Libraries* and is reused either in the *FAST Simulator* and in the *FAST Supervisor Agent*. In Section 3.3 the *FAST Simulator* is described and in Section 3.4 it is explained how the implementation of the process supervision is based in a software agent.

3.1 The FAST Process Definition File

The Process Definition File provides the process description in XML. The eXtensible Markup Language allows the representation of entities, entity properties and entity attributes in a structured form both human and machine readable. In this way, components, relations and process variables are represented as XML data structures. The information in these structures is obtained from the component physical properties (e.g. the device data sheet) and from the P&ID which describes the process topology. During the design process, the process engineer will select and represent the process components in the P&ID diagram according to the process objectives and specifications. In this design phase, is when it is more important to obtain information about the process redundancies. Therefore, *FAST* can be used stand-alone, that is, without being connected to the real process but just providing the Process Definition File (PDL), in order to analyze the FDI capabilities of the process under design. In effect, the process engineer will be able to perform what-if analysis and identify if adding or relocating process components the reliability can be increased. In [Samantaray, 2004], it is illustrated how using a software tool and through the systematic structural analysis of a process, it is possible to improve the sensor placement in order to increase the observability and therefore the capacity for fault detection and isolation. The Process Definition FiLe (PDL) defines the process topology and composition by identifying the list of process variables $\mathcal{V} = \{v_k\}$, process components $\mathcal{C} = \{c_m\}$ and component relations $\mathcal{R} = \{r_n\}$ (see Listing 4.1). Each relation r_n corresponds to a mathematical relation linking several process variables: $r_n : f(v_k)$. A relation can only belong to one component, that is, the association between a relation and a component is unique, although a component normally has associated several relations.

3.1.1 Process Variables

The process inputs and outputs are represented in *FAST* as Process Variables $\mathcal{V} = \{v_k\}$. For each component, it is possible to identify a set of relations between these inputs and outputs. The Process Variables define the connection between components since normally more than one component will have a relation where the same Process Variable is involved. Process Variables can be measured ($\mathcal{V}_m$) or not measured ($\mathcal{V}_u$), $\mathcal{V} = \mathcal{V}_m \cup \mathcal{V}_u$. A measured process variable is a process variable for which exist a sensor providing

LISTING 3.1: Process description in an XML file

```
<ProcessVars>
   <ProcessVar Name="Flow␣Input" Symbol="Qp" Init="0"/>
   <ProcessVar Name="Flow␣Input␣Measured" Symbol="mQp"/>
...
</ProcessVars>

<Components>
   <Component Name="Tank1"/>
   <Component Name="Tank2"/>
...
</Components>

<Relations>
    <Relation Name="R01" Type="Sensor">
      <Component>Level Sensor1</Component>
      <ProcessVar>H1</ProcessVar>
      <ProcessVarMeasured>mH1</ProcessVarMeasured>
   </Relation>
...
</Relations>
```

the variable value. Unmeasured variable values do not have a sensor available and their values shall be obtained from the analytical relations.

In the PDL, a process variable has the following attributes:

- Name: Descriptive name of the process variable (f.i.: “Flow Input”)
- Symbol: Identifier of the process variable (f.i.: “Qp”)
- Init: Initial value used for simulation
- Units: Descriptive name to indicate the physical units
- Tag: In case this process variable corresponds to a tag in the OPC Server, this identifier links the process variable to this tag.

Example: <ProcessVar Name="Flow Input" Symbol="Qp" Init="0" Units="m3/s" Tag="FTE_1001_01"/>

3.1.2 Components

A component in *FAST* is the minimum unit to identify in case of a fault. The correct definition of components is of key importance since it will drive all the diagnostic possibilities. A first approach is to identify as a component each physical entity which has a specific function in the process although *FAST* allows to extend this definition to process objectives which can be expressed as relations and therefore with the possibility of being implemented and used in the tool. A component shall have at least one relation, involving one or more process variables.

In the PDL, a component has the following attributes:

- Name: Descriptive name of the component (f.i.: "Tank1")
- Tag_Error: Identifier of the tag used to communicate the status of the component (OK/Error) to the OPC Server/SCADA.

Example: <Component Name="Tank1" Tag_Error="TK_1001_01_ERR"/>

3.1.3 Relations

Relations are presenting the model of the component. Relations shall be derived from the component behavior. Usually, relations will be mathematical expressions involving one or more process variables. *FAST* will take into account the causality of the variables participating in a relation when performing the structural analysis, as this causality can be determined at design time just by analyzing the structure of the relation.

In the PDL, a relation has the following attributes:

- Name: Descriptive name of the relation (f.i.: "Pump")
- Component: component to which the relation belongs.
- Process Variables: symbols of the specific process variables participating in the relation.

Relations have some fields dependent on the relation type. Examples of types of relations are: Sensor, Pump, ControllerPID, ControllerOnOff, Valve2Levels, ValveLevel, Derivative, Tank, FlowJunction.

For instance, a process which includes a liquid tank will incorporate the component `Liquid_Tank`. The cylindrical tank has a level sensor in millimeters and one input and one output valves. The following relations associated to the tank will be defined:

1. Cylindrical Tank Relation: defines the relation of the level variation inside the liquid tank with respect to the input and output flows and the tank geometry.
2. Derivative Relation: defines a derivative relation between two process variables. In this case, it will be the variation of the level and the level.
3. Sensor Relation: it will exist for each measured variable. In this case, it will be the liquid level.

For this component, the process variables corresponding to the tank level and input and output flows will be defined. The relations model is implemented into *FAST* and therefore just by indicating that the component `Liquid_Tank` contains this set of Relations with the defined Process Variables, the component model will be instantiated and initialized from the data provided in the PDL. Relation models can be extended to cover a wide range of components and component configurations. As the software is structured in classes, new relations can be added by just creating a new class for every new relation. New relations inherit from the Relation virtual class and must implement the virtual methods which are mandatory to be used by the software model. Mainly, it will implement the expression (analytical, algorithm or even fuzzy) which will relate the indicated process variables and the different explicit forms to obtain the value of each causal process variable (see Appendix B).

The task of grouping and defining the relations associated to a component is very important as we are interested in identifying faults at component level. Eventually, the

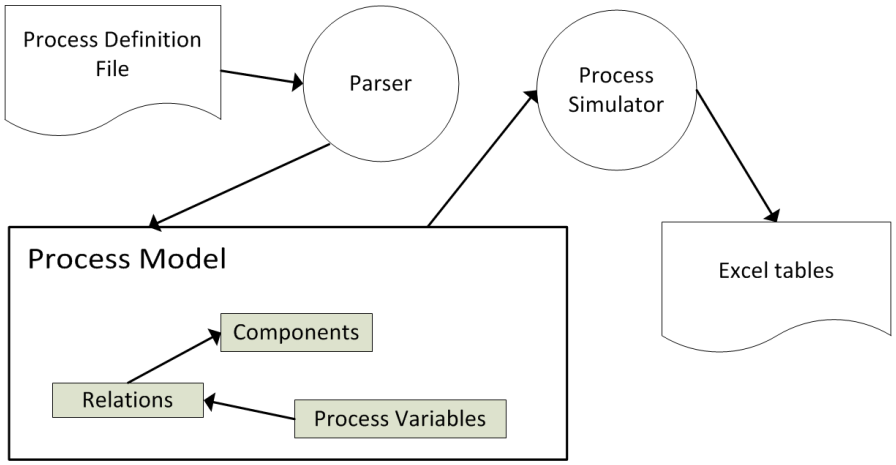


FIGURE 3.1: *FAST Simulator* processes

relations define the model of the component or process objective and deviations from the component model will be identified as faults. Thus, the real behavior measured by the sensors, and the possible analytical redundancy is what will provide the fault detection and isolation capabilities. Every Process Definition File identifies the information related to one process. Therefore, if the plant is formed by several related processes, several files will be generated, one for each process.

3.2 The Process Model

FAST loads the PDL file through the Parser component (see Figure 3.1) generating the Process Model. The Process Model is commonly used in the *FAST* Simulator and the *FAST* Supervisor agent. The PDL is the only input used, besides some ancillary configuration parameters, to generate the Process Model.

The parsing of the PDL provides to the tool the inputs to perform the structural analysis of the process [Maquin and Cocquempot, 1997]. This initialization process will generate the Structural Matrix (M), obtain the analytical redundant relations (ARRs), the residuals and the Fault Signature Matrix $\mathbf{F}$. All this information will form the Process Model (PM).

FAST is able from the PDL to perform the structural analysis, obtain the set of analytical redundant relations $\mathcal{R}_{ARR}$ and the set of residuals $\mathcal{Z}$. The residuals are generated in a way that their evaluation hold near to zero (≈ 0) when the system is working without faults and different ($\neq 0$) if there is any fault in the components that are detectable by means of this residual.

The component models are defined following a similar approach as indicated in [Blanke et al., 2006], where components implement a set of services, use modes and can be aggregated to generate more complex components. In fact, a system goal can be better identified into a process than into a single or aggregated component. Therefore, we defined two type of components, the physical entities of the process and the process objectives. Both can be defined by `Relations` inside a Process.

`Components`, `Relations` and `ProcessVars` are represented as Java classes and compiled into a Java library. For the `Relations`, every relation type is implemented as a derived class. The parent `Relation` class has some generic methods implemented

which provide all common functionality and it is abstract. The Parser module implements the parsing of the Process Definition File and creates the Process abstraction as instances of `Components`, `Relations` and `ProcessVars` related accordingly. The Process Simulator module performs the process simulation generating a set of Excel files reporting; the Structural matrix, the Perfect Matching matrix, the Analytical Redundancy Relations list, the process simulation results and the residuals evaluation results for simulated faults. The tool can be extended by adding more `Components` and its `Relations` as additional Java classes to the library. Several models of the same component can be defined by generating alternative relations and select which ones are the more appropriate.

When *FAST* is used on-line, this software is in fact part of a software agent. The `Components` abstraction form its knowledge base, the agent goals are the to detect and notify process faults and the behavior is implemented through plans. The BDI (Believes, Desires, Intentions) agent structure [Wooldridge, 1992] fits perfectly with the requirements of the automatic process supervision, providing communication, autonomy, reactivity and pro-activity. A software agent is created to wrap every system process. The agents implement the concept of use modes and a set of common services which can be shared among them to provide distributed fault diagnosis capabilities. The agents are executed in the JADEX multi-agent platform [Braubach and Pokahr, 2007].

One of the key skills of *FAST* is the calculation of residuals. From the structural analysis, the ARR set is obtained. Then, from every ARR_i, there is a related set of elementary relations which allows the tool to calculate all process variables participating in the ARR_i. Therefore, in every cycle, *FAST* will acquire the process values for the measured variables and next it will calculate all the residuals by calling for each ARR_i, which are in fact a subset of Relation instances, the method *calcResidual*. This method will obtain from the elementary relations, the value of the unmeasured variables. For every process variable, the method goes through the list of related elementary relations checking if the elementary relation contains this variable and defining a perfect matching with respect to this process variable. The algorithms to obtain the ARR and the Fault Signature Matrix $\mathbf{F}$ are detailed in the Chapter 4.

3.3 The FAST Simulator

During the design phase, the *FAST* Simulator can be used to analyze the process and the fault diagnosis capabilities. The off-line mode allows the engineer to simulate the process and analyzing different configurations and to introduce error conditions evaluating the effects and the diagnoser outputs. The basic products of the *FAST Simulator* are several comma separated files which can be opened in MATLAB or MS Excel with the simulation results (outputs, fault diagnosis, residuals) which can be used to generate graphs and reports to support the design decisions.

The *FAST Simulator* is composed of the two components: the *FAST SimApplication* and the *FAST Libraries*, see the Figure 3.2.

FAST Simulator Application

The *FAST SimApplication* is implemented in Java (tested in a Java Virtual Machine version 7), and therefore it can be executed in any of the supported platforms and operating systems. The computation requirements depend on the number of components

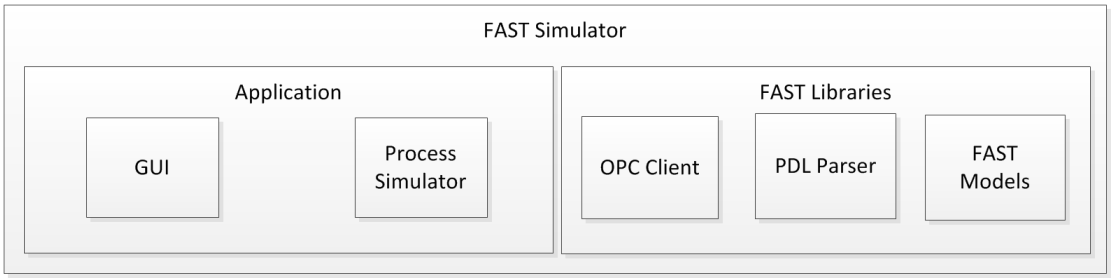


FIGURE 3.2: *FAST Simulator* software components

to be analyzed. The process described in Section 4.4 takes less than 1 second to obtain the structural analysis in a PC with an Intel Core I5. The tool have a simple graphical interface (in English) for loading the PDL and select to start the simulation and a log window. The results are always generated in comma-separated value (CSV) files. These files can be loaded in MATLAB or MS Excel to display the results and generate graphs. The OPC interface enables *FAST* to display the fault detection into the synoptic of an SCADA. The *FAST Simulator* uses the data structures in the Process Model to simulate the process from an initial state by computing the component relations. The tool is able to simulate a fault during a pre-defined time interval and evaluate its effects, calculating the observed fault signature and applying the Fault Signature Matrix to identify the faulty component. The *FAST Simulator*, by using a client OPC interface [Young and Trindade, 2013], is able to feed the process sensors with the simulated values, allowing to the control engineer to rehearse different faults and the system reaction as well as different system configurations. The values are provided to an OPC Server to which the Supervisor Agent can be connected as in a real system.

The *FAST SimApplication* is composed of the following components:

- The Graphical User Interface (GUI) classes: The GUI classes implement the user interface to load the PDL interfacing with the PDL Parser and to start the simulation interfacing with the Process Simulator. The GUI allows the operator to see all loaded components, relations and process variables from the PDL in a two-pane view.

- The Process Simulator: The process simulator is the component which from the Process Model computes the process outputs, calculates and evaluates the residuals, generates the fault vector and in case of simulating a fault, compares the vector with the Fault Signature Matrix to check if the faulty component can be identified. The Process Simulator also interfaces with the OPC Client and if connected to an OPC Server it will provide the simulated values for the relations and in case that the fault can be associated to a component the indication of the fault in this component.

FAST Libraries

The *FAST Libraries* are composed of the following components:

- The OPC Client: The OPC client is the software interface which allows the application to communicate with the OPC Server by sending the tag values corresponding to the relation outputs and the fault indication for the components. The tag need to be defined either in the PDL and in the configuration of the OPC Server.

FIGURE 3.3: *FAST Supervisor Agent* software components

- The PDL Parser: This component is responsible of parsing the PDL file which is an XML. It uses the XML Document Object Model libraries which implement the parsing of generic XML structures. Each component type and relation type has its own function in order to be instantiated correctly in the Process Model. Right after loading the PDL and instantiating the data structures it will call the function to perform the Structural Analysis interfacing with the Process Model classes.
- The Process Model: The Process Model is the set of classes implementing the data structures required to manage the Components, Relations and Process Variables and the Fault Signature Matrix $\mathbf{F}$. Note that after performing the Structural Analysis all relations will be associated between them as elementary relations or redundant relations. Redundant relations will be resolved by substituting all unknown variables by the corresponding elementary relations which can be used to calculate them. Therefore it is not needed to store permanently in the model the Structural Matrix nor the Perfect Matching.

3.4 The FAST Supervisor Agent

The *FAST* functionality is fully deployed once the process has been physically assembled in the plant and is running. In this case, the tool is able to perform the on-line monitoring of the process by just providing the final PDL corresponding to the final process design. The Process Model forms the observation world of a Software Agent, commonly called the agent knowledge-base.

Usually an industrial process is supervised through a SCADA system. The SCADA system interfaces with the Process Logic Controllers (PLCs) from which acquires the process variable values. The SCADA stores the values of the sensors in the process database, where are refreshed periodically. *FAST* will be connected to the SCADA process database through an OPC interface library. Through this library, *FAST* will have access to the values of the process variables obtained from the process database of the SCADA. The *FAST Supervisor Agent* will take these values and calculate the fault vector through the residuals obtained in the initialization process using the OPC Interface.

The *Supervisor Agent* is based on the multi-agent platform JADEX. The agent is the responsible of evaluating on-line the process residuals as defined in the Process Model. These process residuals are evaluated by acquiring the process data on-line via the OPC interface, connected to an OPC server, which at the same time receives the process data from the Process Logic Controllers. The Supervisor Agent is using the same *FAST Libraries* than in the *FAST Simulator* Tool to implement the FDI functionality while the JADEX Libraries will be used to implement the software agent behavior, see Figure 3.3.

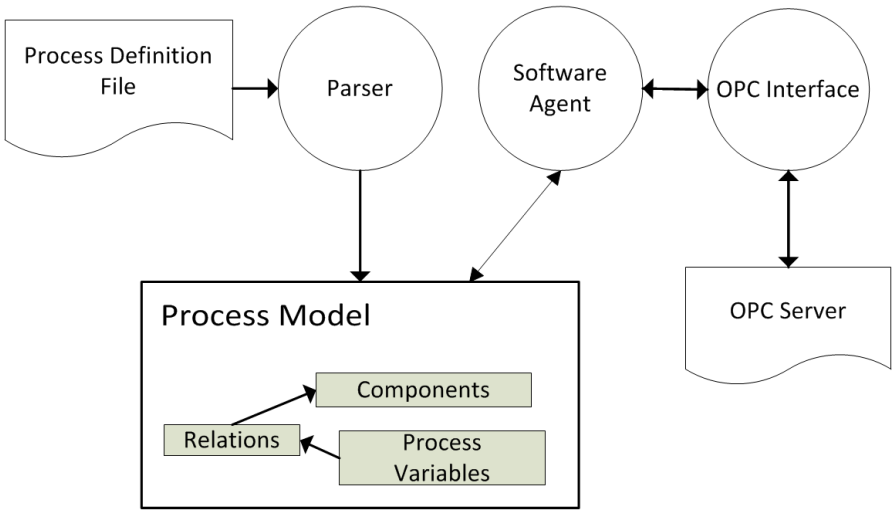


FIGURE 3.4: *FAST Supervisor Agent* processes

In addition, the Supervisor Agent will have as well the capability of being deployed as a multi-agent system and implement the FDI of a process in a distributed manner.

The *Supervisor Agent* is the component responsible of the on-line evaluation of faults and the decision-making part. By using the OPC Interface, *FAST* is able to connect to a process in real-time. The Supervisor Agent will implement a Knowledge Base with the process variables, refreshing periodically their values and feeding the Process Model from them. In the presence of a fault, the Supervisor Agent will activate a Plan to notify the fault, also via the OPC Interface.

In JADEX, an agent is configured by the Agent Definition File, which is an XML file which defines the beliefs, goals, events and plans (see A). The plans are implemented as Java classes which are instantiated at run-time as needed to achieve the agent objectives. In the same way, the beliefs are implemented as a Java class containing the data structures which will compose the Knowledge Base. Several plans can coexist to achieve an objective and reasoning algorithms can be introduced to select the appropriate plan. An scheme of the agent process to select the action to be executed and the execution effects is presented in Figure 3.5. JADEX simplifies the implementation of agents and the capability of handling inter-agent messages. Agent messages are handled as new objectives by applying the corresponding reactive plan. In the case of the *FAST Supervisor Agent*, the software agent will continuously monitor the process with the objective of detecting faults. The agents follow the Believe, Desire, Intention (BDI) architecture, which is composed by the following parts (see Section 2.3.4 for an extended explanation of the generic BDI agents architecture):

- *Knowledge Base* (B): contains all the information needed by the agent to perform its function. It is continuously refreshed and changes in elements can trigger events.
- *Objectives* (Γ): define the set of purposes of the agent which it should autonomously reach.
- *Plans* (Π): define the set of procedures which are available to the agent to reach the objectives.

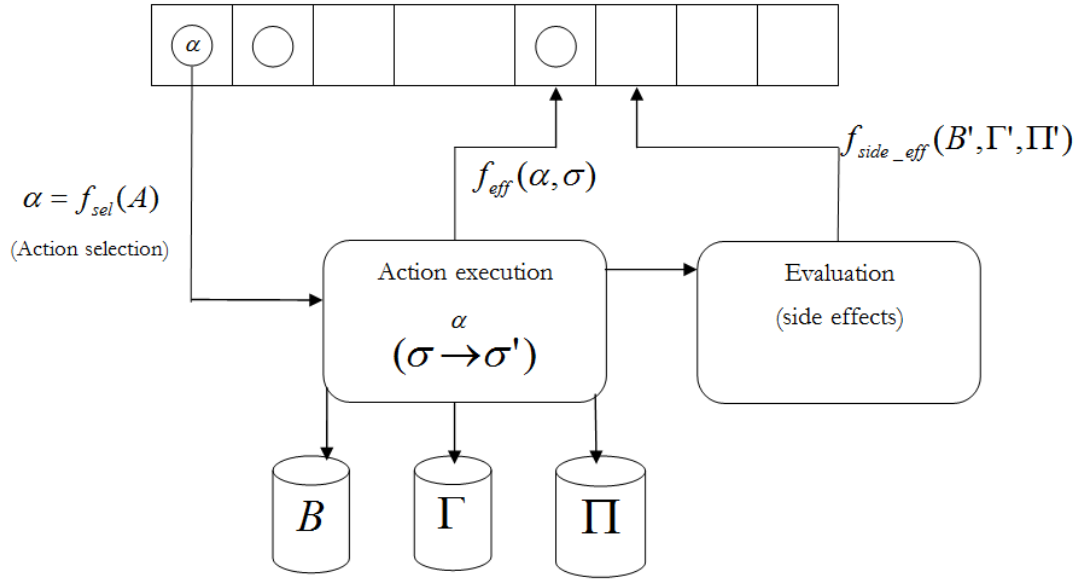


FIGURE 3.5: Supervisor Agent action scheduler

- *Agenda* (A): is the sequence of actions scheduled by the agent. It is continuously changing due to the generation of events, which create new actions and force rescheduling for actions already in the agenda.

A software agent can be seen as a state machine with an indefinite number of states. Each state σ is determined by the tuple $\sigma =< B, \Gamma, \Pi, A >$. Every action executed by the agent will generate a new tuple $\sigma' =< B', \Gamma', \Pi', A' >$, since each of these elements can be affected. Actions are generated as a result of an internal event or by receiving an external event, normally from another agent. The fact of having objectives and plans, although it might seem redundant, allows the agent to apply different plans to achieve the same objectives and have a deliberative process to evaluate which plan is the optimal one to be applied according to the current conditions. For the most simple implementation of our Supervision Agent, there will be a single matching between objectives and plans.

In the case of a single Supervisor Agent, just diagnosing local components, the set of plans is defined as: $\Pi = \{$Calculate_Residuals_Plan, Send_Fault_Plan$\}$. That is, this agent will execute the plan Calculate_Residuals_Plan to achieve the goal Detect_Faults_Goal and the plan Send_Faults_Plan to achieve the goal Notify_Faults respectively. The functionality working in a multi-agent distributed deployment is explained in Chapter 5.

Chapter 4

FAST Structural Analysis

The structural analysis is widely documented in [Blanke et al., 2006] and there are several examples in the literature [Staroswiecki and Declerck, 1989; Maquin and Cocquempot, 1997] which illustrate the way that the analytical redundant relations can be calculated. The algorithms describe how to perform such analysis in an analytical way [Izadi-Zamanabadi and Staroswiecki, 2000] or from bond graphs [Merzouki et al., 2013]. That is, involving symbolic transformations, graphical interpretation or manual derivation of the analytical redundant relations starting from the process model and usually restricted to specific cases. Some tools have already been proposed [Blanke and Lorentzen, 2006; Bouamama et al., 2005] but without covering the full cycle that goes from the analysis up to the on-line process interaction.

The structure of this chapter is as follows: in Section 4.1 the representation in *FAST* of a process is exposed. The Section 4.2 introduces the algorithms implemented in *FAST* to perform the structural analysis. In Section 4.3, the residuals evaluation algorithm is presented. Finally in Section 4.4, the results of verifying the *FAST* tool with the two-tank system case study are presented, as it is a case very well documented in the literature and models of the components and even simulators are available to compare the results.

4.1 Process Description

In order to apply the FAST tool, a process should be described providing the set of process variables ($\mathcal{V}$), the set of components ($\mathcal{C}$), and the association of the process variables to the set of elementary relations ($\mathcal{R}$) associated to the process components. Then, this process description is translated to an XML file constituting the Process Definition File (PDL). This is a possible solution to the requirement identified by Blanke in [Blanke et al., 2006] of having a language for describing systems and their components in a generic form.

FAST generates an abstraction of the process by parsing the PDL, which declares for each process the three groups of items of interest: components $\mathcal{C}$, relations $\mathcal{R}$ and process variables $\mathcal{V}$ (see Listing 4.1). Declaring the `ProcessVars`, *FAST* is able to identify the set of process variables $\mathcal{V}$, which contain both, m measured ($\mathcal{V}_m$) and u not measured ($\mathcal{V}_u$) variables, i.e. $\mathcal{V} = \mathcal{V}_m \cup \mathcal{V}_u$, $l = m + u$. The declaration of the `Components` will allow to generate the component oriented fault signature matrix that will constitute the core of the fault diagnosis algorithm and will allow to identify which component is faulty. All the declared `Relations`, $\langle r_k \rangle \in \mathcal{R}$, will be associated to a component, $c_i \in \mathcal{C}$. FAST implements the model of each component by means of a set relations. Therefore, the relations group will provide the association of the `ProcessVars` to the `Relations` and the `Relations` to `Components`. For instance, in Listing 4.1 the relation `R01` is declared, which corresponds to the level sensor of the

LISTING 4.1: Process description in an XML file

```
<ProcessVars>
        <ProcessVar Name="Flow␣Input" Symbol="Qp" Init="0"/>
        <ProcessVar Name="Flow␣Input␣Measured" Symbol="mQp"/>
...
</ProcessVars>

<Components>
    <Component Name="Tank1"/>
    <Component Name="Tank2"/>
...
</Components>

<Relations>
     <Relation Name="R01" Type="Sensor">
       <Component>Level Sensor1</Component>
       <ProcessVar>H1</ProcessVar>
       <ProcessVarMeasured>mH1</ProcessVarMeasured>
    </Relation>
...
</Relations>
```

tank 1 component (`Tank1`) using the Process Variable H_1 (`H1`) which is representing the actual level of the tank and the measured variable H_{m1} (`mH1`) which is representing the measured value provided by the sensor. *FAST* already knows the sensor model which is instantiated as $\langle r_1 \rangle : H_1 = H_{m1}$ (noise, precision and other parameters are not represented for simplicity).

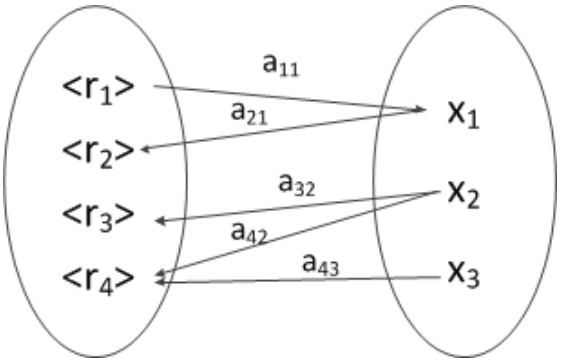
4.2 FAST implementation of the Structural Analysis

In this Section, we are going to introduce the algorithm to perform the structural analysis implemented in *FAST*. The algorithm generates: (i) the structural matrix $\mathbf{M}$, (ii) the perfect matching set ($\mathcal{A}_{pm}$), (iii) the set of analytical redundant relations ($\mathcal{R}_{ARR}$), and (iv) the set of residuals $\mathcal{Z}$ and the fault signature matrix $\mathbf{F}$. The algorithm starts from the representation of the system as a bipartite graph (see Figure 4.1) between the process variables $\mathcal{V}$ and the component relations $\mathcal{R}$.

Given the system description by the tuple $< \mathcal{C}, \mathcal{R}, \mathcal{V} >$, as introduced in Section 4.1, we define $\mathcal{A} := \{a_{ij} | \exists\, a_{ij} : \langle r_i \rangle \rightarrow x_j\}$ as the set of arcs a_{ij} between a relation $\langle r_i \rangle \in \mathcal{R}$ and a process variable $x_j \in \mathcal{V}$, where the arcs are also represented as $a_{ij}(r_i, x_j)$. Defining k as the number of identified relations and l the number of variables, the structural matrix $\mathbf{M}$ will have k rows and l columns. The *Algorithm* 1 builds the structural matrix $\mathbf{M}$ by considering that for each element of the matrix $m_{ij} \in \mathbf{M}$, $m_{ij} = 1$ if $\exists a_{ij} \in \mathcal{A}$ and $m_{ij} = 0$, otherwise.

The Perfect Matching (PM) algorithm identifies the set of arcs $\mathcal{A}_{pm} \subset \mathcal{A}$ which are elementary, that is the set of arcs which links uniquely all unmeasured variables to relations [Blanke et al., 2006]. The PM algorithm is based on initializing $\mathcal{A}_{pm}$ with arcs linking relations corresponding to sensors. The complete PM procedure is presented in

FIGURE 4.1: Bipartite graph example



1: **M:=0**
2: **for** $i = 1$ to k **do**
3: **for** $j = 1$ to l **do**
4: **if** $\exists\, a_{ij} : \langle r_i \rangle \rightarrow x_j$ **then**
5: m_{ij}=1
6: **end if**
7: **end for**
8: **end for**

Algorithm 1: Structural matrix algorithm

Algorithm 2, where it is considered that the first m variables are the measured ones and the last u variables are unmeasured. Similarly, the first s out of l relations correspond to sensors.

1: $\mathcal{A}_{pm} = \{a_{ij} | a_{ij}(r_i) \rightarrow x_j \wedge \langle r_i \rangle \in \mathcal{S}\}$ {$\mathcal{A}_{pm}$ is initialised with sensor relations}
2: **for** $i = s + 1$ to k **do**
3: **for** $j = m + 1$ to l **do**
4: **if** $a_{ij}(x_j) \notin \mathcal{A}_{pm}$ and $a_{ij}(r_i) \notin \mathcal{A}_{pm}$ and $\langle r_i \rangle \rightarrow x_j$ can be explicit **then**
5: $\mathcal{A}_{pm}$:=$\mathcal{A}_{pm} \cup \{a_{ij}\}$
6: **end if**
7: **end for**
8: **end for**

Algorithm 2: Perfect Matching algorithm

Typically the perfect matching matrix $\mathbf{P}$ is generated from the structural matrix, using the perfect matching algorithm, and it is represented in the same way than $\mathbf{M}$ but emphasizing every element indicated as $m_{ij} = 1$ corresponding to an arc a_{ij} which belongs to the $\mathcal{A}_{pm}$ set. It is said that "the perfect matching is reached" if the number of arcs in $\mathcal{A}_{pm}$ is the same than the number of not measured variables, $\#\mathcal{A}_{pm} = \#\mathcal{V}_u$. In this case, it is possible to identify which relations are redundant, that is, without any arc belonging to the perfect matching set.

The relations not employed by arcs in the perfect matching set, that is, non elementary relations, are the analytical redundant relations $\mathcal{R}_{ARR}$. From these relations it is possible to construct a set of equations $\mathcal{Z} = \{z_1, ..., z_n\}$ with $\#\mathcal{R}_{ARR} = \#\mathcal{Z}$, which are commonly known as residuals, by writing each redundant relation with all the not measured variables substituted by the corresponding elementary relations and with all terms grouped in the left hand side and equal to 0. The *Algorithm* 3 identifies

the set of elementary relations required to calculate the residuals, which can be expressed as: If $\langle r_i \rangle \in \mathcal{R}_{ARR}$ is a redundant relation, for each $\langle r_i \rangle$ there will be a set $\mathcal{R}_{el}(\langle r_i \rangle) := \{\langle r_j \rangle | a_{jk}(r_j) \rightarrow x_k, a_{jk} \in \mathcal{A}_{pm}, i \neq j\}$ of elementary relations which permit the calculation of the unmeasured variables in the analytical redundant relation $\langle r_i \rangle$.

for $i = 1$ to $\#\mathcal{R}$ **do**
  if $\langle r_i \rangle \notin \mathcal{A}_{pm}$ **then**
    for $j = 1$ to n **do**
      Determine_Relations($\langle r_i \rangle$, $a_{ij}(x_j)$)
    end for
  end if
end for

Rec_Function{Determine_Relations}{$\langle r_i \rangle, a_{ij}(x_j)$}
for $k = 1$ to $\#\mathcal{A}_{pm}$ **do**
  for each $a_{kj} \in \mathcal{A}_{pm}$ **do**
    if $a_{kj}(x_j) = a_{ij}(x_j) \wedge \langle r_k \rangle \notin \mathcal{R}_{ARR}$ **then**
      $\mathcal{R}_{el}(\langle r_i \rangle) := \mathcal{R}_{el}(\langle r_i \rangle) \cup \langle r_k \rangle$
      for $k = 1$ to n **do**
        Determine_Relations($\langle r_i \rangle$, $a_{kj}(x_j)$)
      end for
    end if
  end for
end for
EndFunction

Algorithm 3: Determining the Redundant Relations

If there is any deviation in the measured values which do not match the resulting residuals model $\mathcal{Z}$, the residuals will deviate from 0. Therefore, by comparing these residuals with a threshold it is possible to identify that there is a fault. We define the fault vector $\mathbf{f} = (f_1, .., f_n)^T$ as the bits' vector resulting of comparing all residuals with the thresholds and putting a 1 if the residual value goes over the threshold and a 0 otherwise.

The Fault Signature Matrix **F** will then be constructed associating a component to a residual when a relation belonging to that component participates in the calculation of the residual. That is, in this matrix, the columns correspond to the components and the rows to the residuals. When a component has an associated relation which participates in the calculation of the residual, we put a 1 in the corresponding row for that residual and a 0 otherwise. To identify that a component is faulty, we compare the inconsistent residuals indicated as a 1 in the fault vector with the Fault Signature Matrix by identifying which components are sensitive to that residual. The matching cannot be implemented comparing directly the fault vector with the column corresponding to a component since a consistent residual is not conclusive. The activation of a residual depends on its sensitivity to the fault and the fault magnitude. The consequence is that more than one component can be identified as faulty. The selection is based on comparing the obtained inconsistent residuals with respect to the expected for each component and take the best matching. However, if there are components with very similar fault signatures in some cases a single identification will not be possible.

FAST is able to perform all the presented analysis and, if working on-line, calculate

the residuals continuously, generate the fault vector and compare it with the Fault Signature Matrix. In this way, the tool is able to detect for some faults if a component of the process has a problem. When connected to a SCADA, the tool is able to communicate via OPC the fault to the SCADA system and the SCADA can display this information in the synoptics. The tool fully deployed is able to monitor multiple processes and fault diagnosis can be also coordinated between them. In Section 4.4, the algorithms are presented in a simple example which will illustrate how all the analysis is executed and which results are generated by *FAST* when working off-line.

4.3 Evaluating the Residuals

The evaluation process of the residuals to obtain the fault vector has a direct impact on the diagnosis of the system. The noise, the quality of the models, the quality of the signals will affect the residual evaluation. In *FAST*, an adaptive threshold algorithm has been developed to determine if a residual is 'positive', that is, the residual it is indicating a fault, or 'negative' meaning the a fault cannot be detected for this residual. This algorithm takes into account the existence of noise in the measurements. For each residual a window of N samples is defined to calculate the residual standard deviation:

$$\sigma = \frac{1}{N-1}\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})} \tag{4.1}$$

The window works backwards in a sliding window manner, that is, the variance is calculated from past samples and the current sample is compared. In order to avoid false positives caused by spikes or artifacts, a number of consecutive samples are compared. The set of samples $s = \{s_0, \ldots, s_m\}$ is compared with the average signal plus/minus the standard deviation with a safety factor $(5 \cdot \sigma)$. When the signal goes over or under the deviation the residual is signaled 'positive' indicating that there is a potential fault. Once the residual is signaled 'positive', if the signal returns to a nominal value, after the transition from the deviation mark, the system is reset and a new delay is applied in order not to include the faulty signal in the calculation of the margins for the following samples. See Figure 4.2.

The fault signature is then extracted from the residuals which are signaling 'positive'. The fault vector is composed by each residual, indicating 1 if there is a fault signaled positive or 0 if not. Each residual is compared for each component in the fault signature matrix. If there is a 1 in the fault signature matrix for that residual, which indicates the residual participates in the component function, and this residual is indicating a fault, the component error index is incremented, and it is decremented if there is no error. In addition, if there is a 0 in the fault signature for that residual/component and the residual is in error, the error index for that component is decremented. This increases the distance between two different fault vectors for each component. The importance is given when the fault vector has a "1" more than when has a "0". Since a "1" is a strong indication that the fault is detected, while a "0" can be a result of weak fault detection, i.e., the residuals are not able to capture the fault while the fault is present.

FAST has a parameter which defines how aggressive will be the faulty component identification. There are several possibilities in presence of an inconsistent residual:

- Pessimistic approach: only identify that a component is faulty if there is only a single component with the maximum score an this score is maximum with respect to the fault vector.

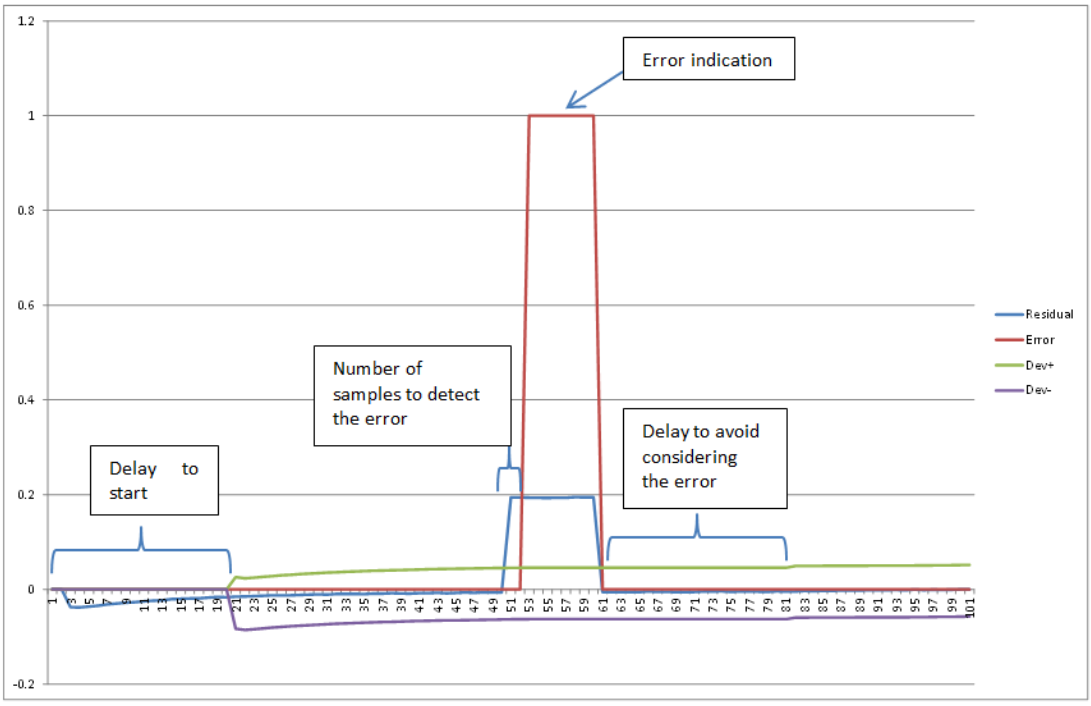


FIGURE 4.2: Residual evaluation

- Medium approach: identify the faulty component if there is only a single component with a maximum score even it is not the maximum for the given fault vector.
- Optimistic approach: identify all faulty components which have a maximum score even if it is not the maximum for the given fault.

Depending on the utilization of the fault information, the most pessimistic or the most optimistic approach can be chosen. For instance if *FAST* is only providing information to the plant operator of the faulty component the optimistic approach could be useful even if more than one component is signaled, however if the information is to be used for automatic reconfiguration of the system the pessimistic approach would be more suitable.

4.4 Case Study: The Two-Tank System

The two-tanks system is proposed to verify the Structural Analysis algorithms implemented in *FAST* as they are commonly used to evaluate the FDI analysis, as for instance in [Ould Bouamama et al., 2001].

In the two-tank system (see Figure 4.3), we identify the following variables which are measured and therefore known:

$$\mathcal{V}_m = \{Q_{mp}, Q_{m12}, Q_{m0}, H_{m1}, H_{m2}, U_{mp}, U_{mb}, U_{m0}\}$$

The prefix m is added to the variable subindex to note that these variables are measured. In addition, we can identify the following components: C = {pump, flow sensor$_1$, flow sensor$_2$, level sensor$_1$, level sensor$_2$, valve$_1$, valve$_2$, PI-controller, on-off controller, tank$_1$, tank$_2$}.

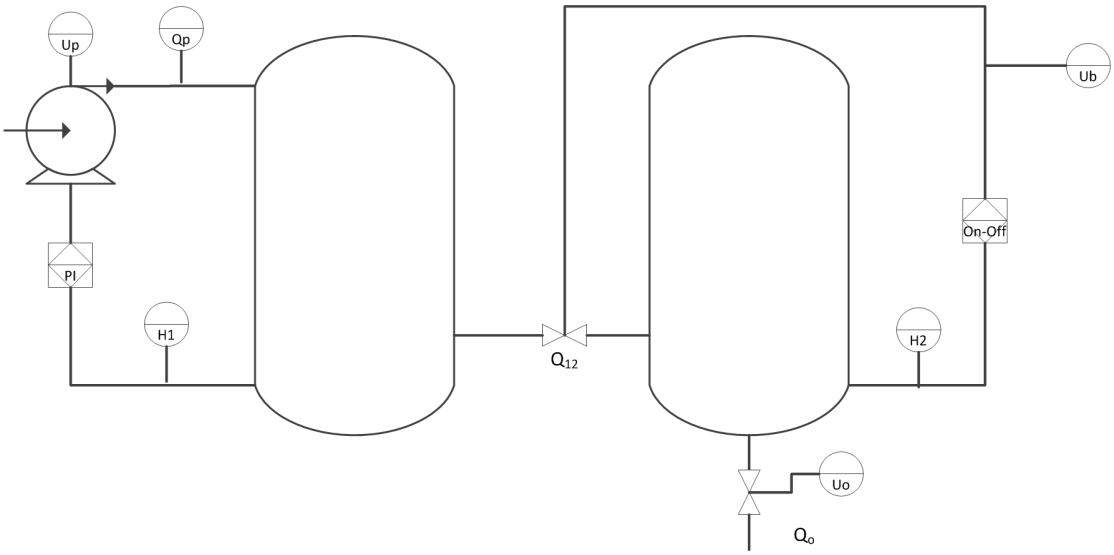


FIGURE 4.3: The two tanks system

For these components, the following relations are derived in a simplified form for sake of clarity, without noise and without additional constraints:

1. Pump
$$Q = a \cdot U_c + b \tag{4.2}$$
considering the most simple case as a linear relation between the output flow and the control signal.
$$U_c = U_{mc} \tag{4.3}$$

2. Tank. Defining Q_i for input flow, Q_o for output flow, and H level,
$$\frac{dH}{dt} = \frac{1}{A}\left(Q_i(t) - Q_o(t)\right) \tag{4.4}$$
$$\dot{H} = \frac{dH}{dt} \tag{4.5}$$

3. Flow sensor.
$$Q_m = Q \tag{4.6}$$
without considering error nor adaptation to the measure range of the sensor.

4. Level sensor.
$$H_m = H \tag{4.7}$$
without considering error nor adaptation to the measure range of the sensor.

5. Valve.
$$Q_o = C_v \cdot \sqrt{H} \tag{4.8}$$
being C_v the valve hydraulic coefficient.

6. PI controller.
$$U_c = K_{pi}(H_c - h_m(t)) + K_{ii}\int (H_c - h_m(t))dt \tag{4.9}$$

7. On-off controller.

$$U_b = \begin{cases} 1 & \text{if } x \geq P \\ 0 & \text{otherwise.} \end{cases} \tag{4.10}$$

with x as the tank level and P as input parameter indicating the tank level set-point.

FAST implements the components in a way that just reading the component list and the relation section from the PDL, all relations are initialised according to the provided data. According to that, the process variables need to be expanded with the variables coming from the analytical relations which are unmeasured:

$$\mathcal{V}_u = \{Q_p, Q_{12}, H_1, \dot{H}_1, H_2, \dot{H}_2, U_p, U_b\}$$

In the PDL, the process variables associated to each relation are defined by indicating the symbol of the process variable. The resulting relations are listed in Table 4.1. When the PDL is loaded into the *FAST* tool, the structural matrix, the perfect matching matrix and the residuals are automatically calculated. For the example, applying *A*lgorithm 1, we obtain the structural matrix. This Table is automatically generated by the *FAST* tool in a comma separated values (csv) file which can be directly transferred into an *Excel* file.

The software *FAST* also generates a table indicating which relations form the perfect matching according to the *A*lgorithm 2 adding an asterisk '*' to the '1' as indicated in Table 4.2. The Structural Matrix is generated in a file which can be also directly imported into Excel.

TABLE 4.1: Component relations

Relation	Type	Component	Variables	Description
$\langle r_1 \rangle$	Sensor	Level Sensor 1	H_{m1}, H_1	Level sensor tank 1
$\langle r_2 \rangle$	Sensor	Level Sensor 2	H_{m2}, H_2	Level sensor tank 2
$\langle r_3 \rangle$	Sensor	Flow Sensor	Q_{mp}, Q_p	Flow sensor at pump output
$\langle r_4 \rangle$	Sensor	Pump	U_{mp}, U_p	Pump control signal
$\langle r_5 \rangle$	Sensor	Valve 1	U_{mb}, U_b	Two tanks linking valve control signal
$\langle r_6 \rangle$	Sensor	Valve 2	U_{mo}, U_o	Tank 2 output valve control signal
$\langle r_7 \rangle$	Pump	Pump	Q_p, U_p	Pump action
$\langle r_8 \rangle$	PI Controller	PI Controller	H_{m1}, U_p	PI controller
$\langle r_9 \rangle$	Valve2Levels	Valve Tanks	Q_{12}, H_1, H_2, U_b	Calculated flow output from the valve linking the two tanks
$\langle r_{10} \rangle$	On-Off Controller	On-Off Controller	H_{m2}, U_b	Controller opening/closing the linking the two tanks
$\langle r_{11} \rangle$	Valve Level	Valve 2	Q_o, H_2, U_o	Calculated flow output from the tank 2 output valve
$\langle r_{12} \rangle$	Tank	Tank 1	$Q_p, Q_{12}, H_{m1}, \dot{H}_1$	Relation between flow input/output and tank 1 level
$\langle r_{13} \rangle$	Tank	Tank 2	$Q_o, Q_{12}, H_{m2}, \dot{H}_2$	Relation between flow input/output and tank 2 level
$\langle r_{14} \rangle$	Derivative	Tank 1	$H_1, \dot{H}_1$	Derivative relation in tank 1
$\langle r_{15} \rangle$	Derivative	Tank 2	$H_2, \dot{H}_2$	Derivative relation in tank 2

TABLE 4.2: Structural Matrix indicating the Perfect Matching with '*'

Relations	U_{mo}	Q_{mp}	H_{m1}	H_{m2}	U_{mp}	U_{mb}	U_o	U_p	Q_p	Q_o	Q_{12}	$\dot{H}_1$	U_b	$\dot{H}_2$	H_2	H_1
$\langle r_1 \rangle$	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1*
$\langle r_2 \rangle$	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1*	0
$\langle r_3 \rangle$	0	1	0	0	0	0	0	0	1*	0	0	0	0	0	0	0
$\langle r_4 \rangle$	0	0	0	0	1	0	0	1*	0	0	0	0	0	0	0	0
$\langle r_5 \rangle$	0	0	0	0	0	1	0	0	0	0	0	0	1*	0	0	0
$\langle r_6 \rangle$	1	0	0	0	0	0	1*	0	0	0	0	0	0	0	0	0
$\langle r_7 \rangle$	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
$\langle r_8 \rangle$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
$\langle r_9 \rangle$	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1
$\langle r_{10} \rangle$	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
$\langle r_{11} \rangle$	0	0	0	0	0	0	1	0	0	1*	0	0	0	0	1	0
$\langle r_{12} \rangle$	0	0	0	0	0	0	0	0	1	0	1	1*	0	0	0	0
$\langle r_{13} \rangle$	0	0	0	0	0	0	0	0	0	1	1*	0	0	1	0	0
$\langle r_{14} \rangle$	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
$\langle r_{15} \rangle$	0	0	0	0	0	0	0	0	0	0	0	0	0	1*	1	0

The *FAST* tool generates the analytical redundant relations in another file, identifying which of the primary relations from the perfect matching can be used to calculate the redundant relations. For the example, the redundant relations obtained are indicated in Table 4.3.

TABLE 4.3: Analytical redundancy relations with the associated elementary relation set

$R_{el}(\langle r_i \rangle)$	Elementary Relations Set
$R_{el}\langle r_7 \rangle$	$\langle r_3 \rangle$, $\langle r_4 \rangle$
$R_{el}\langle r_8 \rangle$	$\langle r_4 \rangle$, $\langle r_1 \rangle$
$R_{el}\langle r_{12} \rangle$	$\langle r_1 \rangle$, $\langle r_{14} \rangle$, $\langle r_3 \rangle$, $\langle r_2 \rangle$, $\langle r_5 \rangle$, $\langle r_9 \rangle$
$R_{el}\langle r_{10} \rangle$	$\langle r_2 \rangle$, $\langle r_5 \rangle$
$R_{el}\langle r_{13} \rangle$	$\langle r_1 \rangle$, $\langle r_{11} \rangle$, $\langle r_2 \rangle$, $\langle r_{15} \rangle$, $\langle r_5 \rangle$, $\langle r_6 \rangle$, $\langle r_9 \rangle$

TABLE 4.4: Analytical redundancy relations with the associated elementary relation set from [Ould Bouamama et al., 2001]

$R_{el}(\langle r_i \rangle)$	Elementary Relations Set
$R_{el}\langle r_{12} \rangle$	$\langle r_9 \rangle$, $\langle r_{14} \rangle$, $\langle r_1 \rangle$, $\langle r_3 \rangle$, $\langle r_2 \rangle$, $\langle r_5 \rangle$
$R_{el}\langle r_{13} \rangle$	$\langle r_2 \rangle$, $\langle r_1 \rangle$, $\langle r_5 \rangle$, $\langle r_6 \rangle$, $\langle r_{11} \rangle$, $\langle r_9 \rangle$, $\langle r_{15} \rangle$
$R_{el}\langle r_8 \rangle$	$\langle r_4 \rangle$, $\langle r_1 \rangle$
$R_{el}\langle r_7 \rangle$	$\langle r_4 \rangle$, $\langle r_3 \rangle$

TABLE 4.5: Fault signature matrix

FSM	Q_{mp}	H_{m1}	H_{m2}	U_{mb}	U_{mp}	Pump	Tank 1	Tank 2	Valve Out	Valve Tanks
$z_1\langle r_7 \rangle$	1	0	0	0	1	1	0	0	0	0
$z_2\langle r_8 \rangle$	0	1	0	0	1	0	0	0	0	0
$z_3\langle r_9 \rangle$	0	0	1	1	0	0	0	0	0	0
$z_4\langle r_{12} \rangle$	1	1	1	1	0	0	1	0	0	1
$z_5\langle r_{13} \rangle$	0	1	1	1	0	0	0	1	1	1

The Table 4.3 can be compared with the results obtained by Bouamama in [Ould Bouamama et al., 2001] indicated in Table 4.4. In this work, the Structural Analysis produced 4 residuals. It is possible to identify quickly that the residuals are the same as the identified by the *FAST* tool. The conclusion is that the tool is providing equivalent results than the previous work which was performed analytically but now applying the *FAST* algorithms in a systematic way.

Furthermore, it is important to remark that *FAST* has found one additional residual considering the redundant relation $< r_{10} >$. This relation is the On-Off controller. The diagnostic information that can be obtained from this relation is only applicable to a restricted number of cases. The On-Off controller is a binary signal depending on the level of the $Tank_2$ and therefore the level process variable H_2 is not causal with respect to this relation and only the control signal U_b can be calculated. Thus only when the

TABLE 4.6: Fault signature matrix from [Ould Bouamama et al., 2001]

FSM	Q_{mp}	H_{m1}	H_{m2}	U_{mb}	U_{mp}	Pump	Tank 1	Tank 2	Valve Tanks
$z_1\langle r_7\rangle$	1	0	0	0	1	1	0	0	0
$z_2\langle r_8\rangle$	0	1	0	0	1	0	0	0	0
$z_3\langle r_{12}\rangle$	1	1	1	1	0	0	1	0	1
$z_4\langle r_{13}\rangle$	0	1	1	1	0	0	0	1	1

On-Off controller will be indicating a valve position which is not in line with the level sensor measured in $Tank_2$ it will be possible to identify a possible fault. However, it is relevant information which could help to better diagnose a fault and therefore is an improvement with respect to the residuals identified in the previous work.

Another difference from the previous work is that in order to use the residual derived from the PI relation and obtain a clear fault indication it is not enough to use the tank level measured variable H_m, as the PI adapts quickly to the error. Instead, the model variable of the tank level H_1 is used. The model variable is obtained from the relation $< r_{12} >$ and $< r_{14} >$.

From Table 4.3 it is very easy to obtain the Fault Signature Matrix associating each component with its fault vector, if the component has a relation which is sensitive to a residual, we place a '1' in the cell of the residual z_n crossing the component or '0' otherwise. The Fault Signature Matrix for the example is indicated in Table 4.5.

If the results of the fault signature are compared with the results obtained in [Ould Bouamama et al., 2001] indicated in Table 4.6, we can see that they are totally equivalent. The only differences are that in the previous work, the Valve Output component and the additional residual coming from the On-Off controller are not considered but for the rest of the components and residuals the fault signatures are identical.

As *FAST* contains the models for each component, it is able to simulate the process dynamics. Therefore, providing the initial values in the PDL, it is possible to simulate the process behavior and generate a file which can also be imported into Excel with the time and the value that each Process Variable takes at that time. In this way, to test the response of the residual calculation, it is possible to indicate that a component has a fault, if the fault is additive, multiplicative or a fixed value and the time interval. For each component, a limited set of faults can be configured depending on the component. This feature is as well coded in the component.

FAST is able to calculate the residuals by simulating the process. The control setting for the $Tank_1$ level is 0.8m while the $Tank_2$ level is controlled by the on/off controller, if the level goes over 0.6m, the valve is closed. The Valve Output is by default closed. For each analytical redundancy relation R_{el} all associated elementary relations are checked. When a relation containing a the process variable is found, it is checked if this process variable can be calculated from that relation, if not, associated relations to this one are also checked recursively, until a relation is found from which the variable value can be obtained.

It is possible to force a component fault in simulation during a defined time interval. In the component section of the PDL, it can be indicated which fault type is forced (additive, multiplicative or a fixed value) and the starting and ending times. In this way, during the simulation, we can obtain the residuals and evaluate the sensitivity of each residual to the indicated fault. For example, to simulate a fault in the $LevelSensor_1$, an additive fault of value 0.2 (range is 0 to 2 liters) is indicated for the time interval from

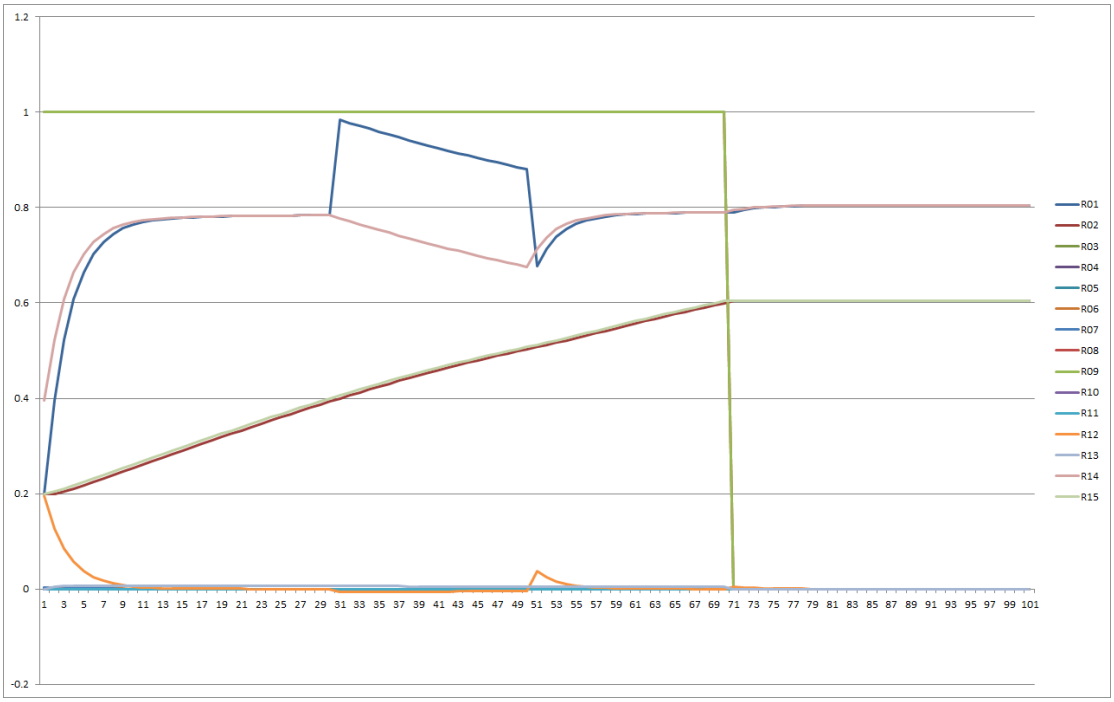


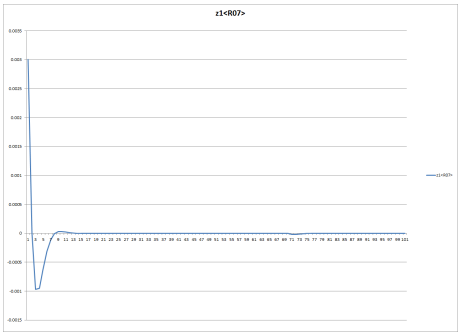
FIGURE 4.4: LevelSensor_1 error simulation

second 30 to second 50. This will add an offset deviation to the $LevelSensor_1$ output of 0.2m, simulating a fault of the sensor as shown in the Figure 4.4.

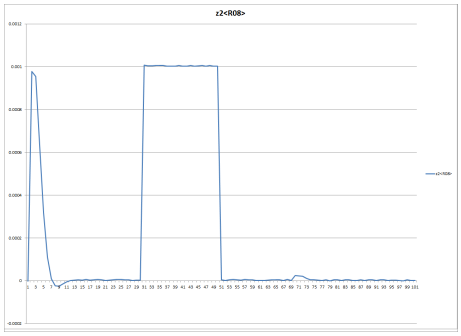
From the simulations it has been verified several already known effects, such as that the noise has a big impact on the fault sensitivity of the residuals. *FAST* also includes the capability of defining the sensor precision by adding a white noise in the range indicated as a parameter in the PDL. For instance, a white noise in the range of 10^{-2} which is a precision of centimeters in the level sensor causes that the offset deviation fault is not detected while if the noise is in the range of 10^{-3} in this case the fault is clearly visible in the residuals.

Figure 4.5 present the residuals calculated by *FAST*. Notice that only the residuals corresponding to the z_2, z_4 and z_5 are sensitive to this fault. This is in accordance with the Fault Signature Matrix presented in Table 4.5. Note that due to the noise for the residual z_5 is more difficult to detect the fault. This is because the effect of the error in the $LevelSensor_1$ has a lower effect in the $Tank_2$. The model is using the derivative to calculate the redundant value of the tank level and the tank level is affected with an increment near to the noise. This is a fact to consider when evaluating the residuals, if the variations of the process variables due to a fault are near the to the noise or to the precision of the sensor it will be difficult to use this residual to identify a fault. The determination of the magnitude of the fault with respect to the sensitivity of a residual is already studied in FDI. The Diagnosis Model Processor [Petti, Klein, and Dhurjati, 1990], for instance, has been proposed which performs an analysis of fault sensitivity with respect to the model in real-time. This method could be incorporated by *FAST* to improve the the fault identification process. Note also that the artifacts on second 68 on the residuals are due to the closure of the valve between the two tanks when the $Tank_2$ reaches the target level.

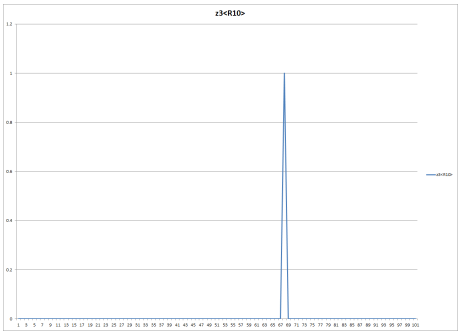
FAST will calculate the fault vector by comparing the residuals with a threshold.



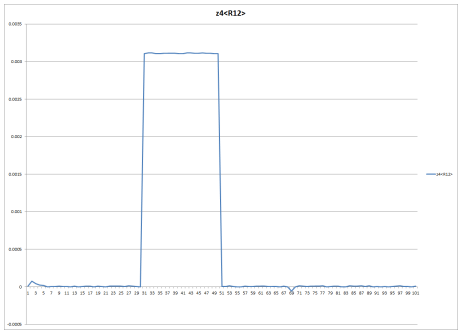
(A) Residual z_1



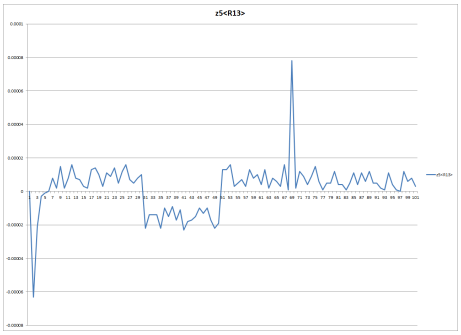
(B) Residual z_2



(C) Residual z_3



(D) Residual z_4



(E) Residual z_5

FIGURE 4.5: $LevelSensor_1$ fault residuals

Each residual with a value over the threshold will imply a '1' in the corresponding position in the vector. For the considered fault scenario, the fault vector is $\mathbf{f_1} = (0, 1, 0, 1, 1)^T$. This vector is compared with the Fault Signature Matrix. In this case, the result matches completely with the column of the Level Sensor_1 fault. Therefore the faulty component is the set $C_{f1} = \{LevelSensor_1\}$.

A second fault is simulated in the $LevelSesor_2$ by forcing a value 0 in the same interval from second 30 to 50. The simulation output is indicated in Figure 4.6. In this case, the residuals sensitive to the fault according to Table 4.5 are z_3, z_4 and z_5. In the Figure 4.7, we can see the residuals as calculated by *FAST* when simulating this fault. The noisy residual is z_4, as it is less sensitive to this fault. The fault vector will be $\mathbf{f_2} = (0, 0, 1, 1, 1)^T$ which according to the Fault Signature Matrix, it matches with the $LevelSensor_2$ and the On-Off controller. Therefore for this fault, the faulty component is the set $C_{f2} = \{LevelSensor_2, On - Off\_controller\}$.

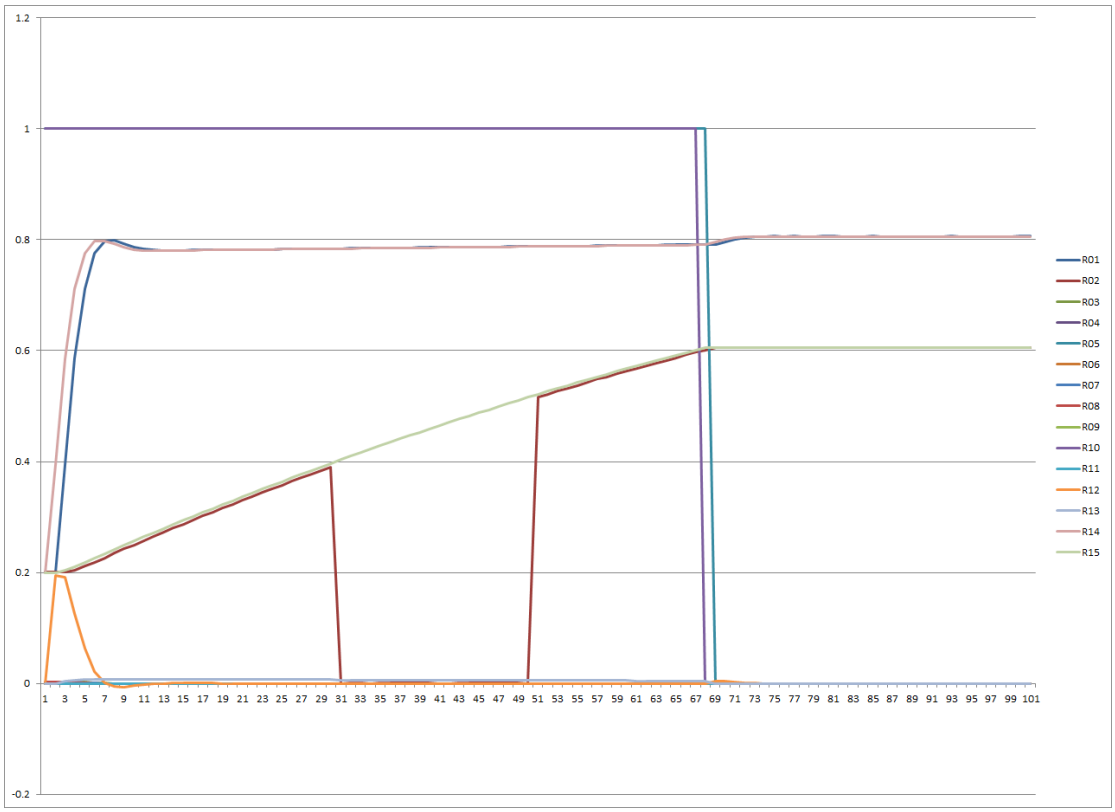
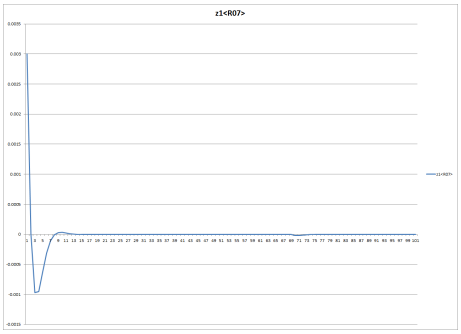
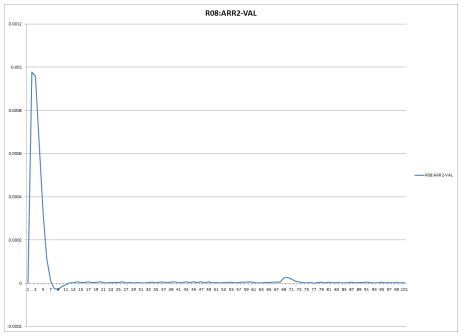


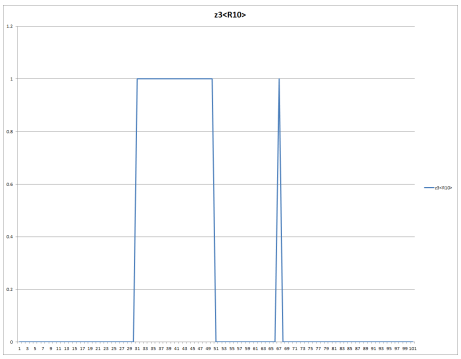
FIGURE 4.6: LevelSensor_2 error simulation



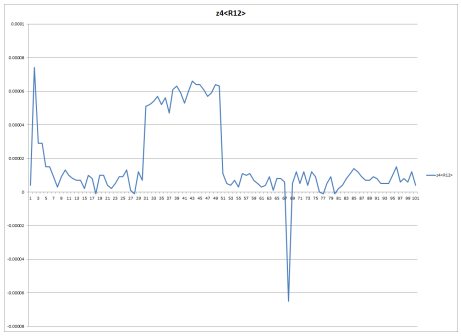
(A) Residual z_1



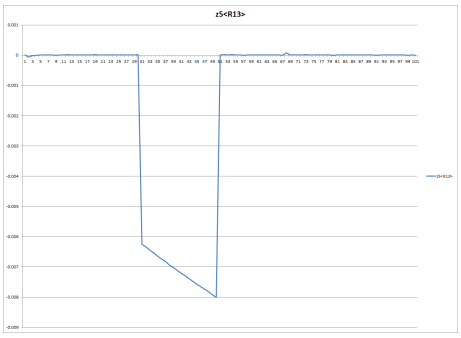
(B) Residual z_2



(C) Residual z_3



(D) Residual z_4



(E) Residual z_5

FIGURE 4.7: $LevelSensor_2$ fault residuals

Chapter 5

FAST Distributed FDI

FAST has been designed as a tool to aid in the fault diagnosis design of complex industrial systems and provide a systematic implementation process. As the tool is able to identify the analytical redundancies early in the design phase, it helps the process engineer to decide the architecture of the system and for instance add or remove sensors or hardware redundancies to increase the FDI capabilities and eventually the reliability of the system. An update of the tool has been implemented to allow partitioning of complex systems into more simple subsystems by following the component coupling criteria. This characteristic will aid the process engineer to distribute among different processors the FDI diagnosers in a complex large scale industrial system. There are several motivations which could request the implementation of distributed FDI. Just to mention a few:

- A very complex system with many different components, which make solving the FDI process a too complex computation task for one single processor.
- A physically distributed system with several processors located in different remote sites but physically interconnected, and therefore sharing FDI process variables.
- A system with well identified subsystems which are controlled by different processors in a process plant.

In addition, the criteria for the partitioning of such systems into subsystems can be also selected by several factors: the proximity between the different components, which could be associated to a cost function; the physical implementation of the connections between components and processors; the physical distribution of the process.

In this chapter, we will propose a method to partition a complex industrial system based on the concept of components coupling with the aim to implement distributed FDI. The coupling between components will be defined in Section 5.1 where also the partitioning of the system and residuals calculation for FDI analysis are introduced. The Section 5.2 explains how by assigning a supervision agent for each subsystem, the complex distributed FDI networked multi-agent system is generated. In Section 5.3, it is explained how the residuals calculation is changed to adapt to the possibilities of sharing the values of the local relations among several connected Supervisor Agents aiding to resolve the distributed diagnosis problem. In Section 5.4, it is briefly discussed how the time affects to the residual calculation, since when the FDI is distributed in a network, the communication delays are not negligible. Finally in Section 5.5, an example based on a water distribution system will be used to illustrate the use of *FAST* for implementing a distributed FDI system.

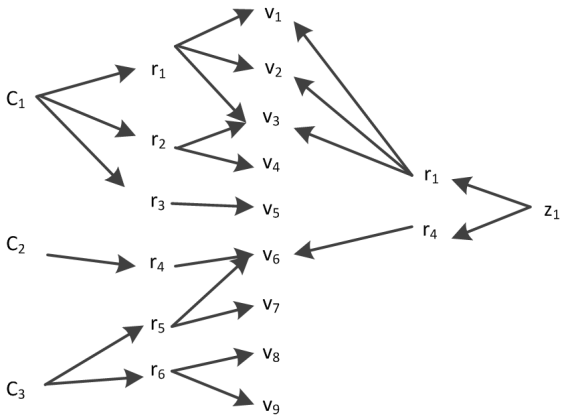


FIGURE 5.1: Component fault detectability definition example

5.1 Partitioning a System

FAST proposes the partitioning of the system from the information provided by the Fault Signature Matrix. The Fault Signature Matrix $\mathbf{F} = \{f_{i,j}\}$ indicates the relation of the residuals (rows) with the process components (columns). A value $f_{i,j} = 1$ indicates that the residual i is sensitive to a fault in the component j. Hence, when the residual evaluation is not zero, we can identify that all components with $f_{i,j} = 1$ in the i-th row of the residual are affected. This association can be established because residuals are calculated by using the redundant relations $\mathcal{R}_{ARR}$ and all relations are associated uniquely to a component as indicated above.

Definition 6 *A fault in the component c_j is defined as* detectable *by means of a residual z_i when $f_{i,j} = 1$ in the Fault Signature Matrix* $\mathbf{F}$.

In *FAST*, the detectability of a component fault by means of a residual is determined by checking whether the elementary relation of the component has been used to generate the analytical redundancy relation associated to the residual. Thus, if $\langle r_i \rangle \in \mathcal{R}_{ARR}$ is a redundant relation and $\mathcal{R}_{el}\langle r_i \rangle$ is the set of elementary relations associated to that relation $\langle r_i \rangle$ and $\mathcal{R}_{comp}\langle c_j \rangle$ is the set of relations associated to a component, a fault in the component c_j is detectable using the residual z_i if there is a process variable v_k which is common to at least in one relation from $\mathcal{R}_{el}\langle r_i \rangle$ and $\mathcal{R}_{comp}\langle c_j \rangle$.

In the Figure 5.1, the components set $\mathcal{C}$=$\{c_1, c_2, c_3\}$, the set of relations $\mathcal{R}$=$\{r_1, ..., r_6\}$ and the set of process variables $\mathcal{V}$=$\{v_1, ..., v_9\}$ can be identified. Let us consider that the structural analysis concluded that the residual z_1 is obtained by combining the elementary relations r_1 and r_4. Then, faults in components c_1, c_2 and c_3 will be detectable by means of this residual, as they share the variables v_1, v_2, v_3 and v_6 which participate in relations r_1, r_2, r_4 and r_5 associated to the these components.

Definition 7 *A component c_j is defined as* coupled with a component c_k, *with respect to a set of residuals $\mathcal{Z}_c = \{z_n, ..., z_l\}$ if faults in both components are detectable in one or more residuals of the set $\mathcal{Z}_c$. Hence,* two components are coupled with respect to the residual $z_i \in Z_c$ *if $f_{i,j} = 1$ and $f_{i,k} = 1$.*

The coupling property could be also quantified. That is, by calculating the Hamming distance $\oplus$ between the fault signatures associated to one or more components with respect to a residual subset, we can determine the coupling level between the

component with respect to this subset. For example, if the Hamming distance of a component fault signature with respect to all faulty components detectable by the residuals from one set is 3 and from another residual subset is 1, we can say that this component is more coupled with respect to the second set than with respect to the first one. Formally, if $\mathbf{f}_c$ is the fault vector of a component and $\mathcal{Z}_c$ is a residual subset, we define the function $\zeta(c, \mathcal{Z}_c)$ as:

$$\zeta(c, \mathcal{Z}_c) = \frac{1}{\frac{1}{n}\sum\{\mathbf{f}_c \oplus \mathbf{f}_j\}} \tag{5.1}$$

being $\mathbf{f}_j, j = 1, \ldots, n$ the fault signatures of the components detectable by the residual subset $\mathcal{Z}_c$, and $\mathbf{f}_c$ the fault signature associated to the component c.

This definition is trivial for two components which are detectable by the same residuals. However, it is more relevant in the case that there is a group of components detectable only by a specific subset of residuals, which means that they are coupled between them but not coupled with another subset. In addition, this definition will help to determine the subsystem to which those components which are coupled to more than one residuals subset will be assigned by quantifying the level of coupling and assigning the components to the subsystem with a higher coupling level.

Definition 8 *Two components are defined as* independent with respect to the set of residuals $\mathcal{Z}_d$ *if they are not coupled with respect all the residuals in this set. That is, a component c_i, where $f_{i,j} = 1$ is independent of a component c_k, if there is not any $f_{i,k} = 1$ for any residual $z_i \in \mathcal{Z}_d$.*

Once the Fault Signature Matrix $\mathbf{F}$ is obtained for a process, it is possible to group the residuals by considering the coupling between components. If we sort the columns (the components) of $\mathbf{F}$ by moving to the left the columns where there is a value "1" in the rows from top to down, and then we sort the rows (the residuals) in the same way by moving up the residuals which have a value "1" from left to right and so on, we will finally obtain the components and residuals sorted in a way that there are components only detectable by a specific subset of residuals (see Algorithm 4). In fact, we are grouping components which are coupled according to *Definition* 2.

Definition 9 *A group of components C_a is defined as* independent *from another group of components C_b if all the components in C_a are independent from any component in C_b. That is, there are no components which are detectable by the residuals in the group C_a and C_b at the same time.*

This definition helps to identify groups of components which are totally independent. These groups do not share any relation or process variable and therefore can be diagnosed locally. All information to implement the FDI diagnoser is self-provided by the components in the group. These groups will originate the candidates to be defined as subsystem.

Definition 10 *A* shared component *is defined as a component which is coupled to two or more components from different independent groups.*

It is easy to deduce that independent component groups could be partitioned into subsystems, and consider only the residuals affecting to those subsystems. However, as the subsystems have relations which share process variables, there will be component faults affecting residuals from more than one subsystem. For these shared components,

it is possible to join them to any of the independent groups to which they are coupled. Nevertheless, we propose to use the coupling level to assign the shared components as per *Definition 2*. We identify the set of components as $\mathcal{C}_a$, and $\mathcal{Z}_a$ is the set of residuals that is able to detect faults in this set of components $\mathcal{C}_a$. On the other hand, we have $\mathcal{C}_b$ and $\mathcal{Z}_b$ corresponding to another set of components detectable by means of a different set of residuals, and the set $\mathcal{C}_a$ is independent from $\mathcal{C}_b$ as per *Definition 4*. We calculate the coupling level of c_s with respect C_a and C_b by solving $\zeta(c_s, \mathcal{Z}_a)$ and $\zeta(c_s, \mathcal{Z}_b)$. Therefore, we will join the shared components to the component group for which the resulting coupling level value is maximum. That is, the sum of the Hamming distances between the shared components and all the components from the independent group is minimum. The fact of using the coupling level results on a definition of partitions which minimizes the interdependencies between subsystems. The diagnoser for subsystems with shared components will calculate the residuals involving these components and will share the value of the process variables to other diagnosers.

1: {Sort columns}
2: **for** $i = 1$ to $\#\mathcal{Z}$ **do**
3: **for** $j = 1$ to $\#\mathcal{C}$ **do**
4: **if** $f_{ij} = 1$ **then**
5: swap($\mathbf{c}_j$, $\mathbf{c}_{j+1}$)
6: **end if**
7: **end for**
8: **end for**
9: {Sort rows}
10: **for** $j = 1$ to $\#\mathcal{C}$ **do**
11: **for** $i = 1$ to $\#\mathcal{Z}$ **do**
12: **if** $f_{ij} = 1$ **then**
13: swap($\mathbf{z}_i$, $\mathbf{z}_{i+1}$)
14: **end if**
15: **end for**
16: **end for**
17: **EndFunction**

Algorithm 4: FDI subsystem partitioning algorithm.

Once the structural analysis of the full process is performed, *FAST* has the capability of pre-processing the Fault Signature Matrix of the full process, identifying the independent and shared component groups and partitioning them into subsystems according to the definitions provided above. Thus, from the set of components $\mathcal{C}$, there will be n subsets of components. Each subsystem will be monitored by a "local" diagnoser which in *FAST* is implemented by a Supervisor Agent. Therefore, *FAST* will create an instance of a Supervisor Agent for each subsystem. When a value of a variable is required to calculate a residual and this variable cannot be obtained from the local set, the Supervisor Agent will request this value to the neighbor Supervisor Agent instances and will use this value to solve the diagnosis, that is, to calculate the observed fault signature.

In the distributed implementation of the fault diagnosis system, not all relations can be solved locally. Therefore, the algorithm needs to take into account if a relation can be solved locally or not, and if it cannot be solved locally request the calculation to a remote Supervisor Agent in which the relation can be solved. All Supervisor Agents

implement a function to check whether they are the "owner" of the relation. A Supervisor Agent is the owner of a relation if the relation is associated to a local component. In case that the agent is not the owner, it will send a message to the Supervisor Agent for which the associated component is local and will wait for the response with the result of the relation calculation. This functionality is implemented by the function *Remote_Calculate* (see Algorithm 5).

Function{Calc_From_Relation}{v_j, $\langle r_i \rangle$}
for each $\langle r_k \rangle \in \mathcal{R}_{el}(\langle r_i \rangle)$ **do**
  if $v_j \in \mathcal{V}_k$ **then**
    if IsLocal($\langle r_k \rangle$) **then**
      $result$=Calculate($\langle r_k \rangle, v_j$)
    else
      $result$=Remote_Calculate($\langle r_k \rangle$, v_j)
    end if
  end if
end for
EndFunction

Algorithm 5: Calculating the residuals in a distributed FDI.

5.2 Distributed Supervisor Agents

There are several network architectures to define a multi-agent system. Although basically they can be split in two groups: the hierarchical distribution, where there is a higher level entity which communicates with the entities of the lower level and provides an encapsulated view to higher level entities, and the decentralized approach (peer to peer) where each entity is at the same level and communicates with the immediate neighbors. In this work, we have chosen the decentralized approach, since it has several well known advantages:

- There exist less implementation dependencies, that is, if the agents act in a hierarchical level the information flow needs to be structured in the same way at each level.
- It is less sensitive to modifications, the agent only know their neighbors and any change beyond its immediate vicinity will not have any effect.
- The computation complexity is maintained linear, each agent will have an equivalent computation complexity while in a hierarchical organization, the upper level entity will require a higher computation complexity depending on the number of child.
- The same agent implementation can be used for the diagnostic of all subsystems.

The hierarchical approach, however, has the advantage that a services directory can be made available and the agents could query the required services from other agents from this directory service in run-time, while in the pure decentralized approach the interfaces with the surrounding agents need to be known in advance at design-time or the agents should use broadcast messages.

In [Console, Picardi, and Theseider Dupré, 2007], an example of architecture is presented; although the authors indicate that the architecture is decentralized, a higher

LISTING 5.1: Agents definition in the PDL file

```
<Agents>
<Agent name="SA1@SRV01"
opc_server="Matrikon.OPC.Simulation.1"/>
<Agent name="SA2@SRV01"
opc_server="Matrikon.OPC.Simulation.1"/>
</Agents>
```

level Supervisor entity is required to coordinate the Local Diagnosers of all subsystems. In *FAST* there is no need of a single Supervision system, since all diagnosers work in the JADEX multi-agent system based on the architecture proposed in [Wooldridge, 1992] and are self sufficient to perform the diagnosis of the corresponding subsystems. However, in order to implement a human-machine interface where the diagnosis results can be provided human readable, *FAST* implements the OPC client interface which is able to communicate with a SCADA system. The SCADA system does not need to be unique, since each Supervisor Agent is configured to communicate to a specific SCADA. In the same way, the Supervisor Agents in *FAST* read the subsystem signals through this OPC interface. The OPC interface requires an OPC Server which implements the physical communications with the controllers and sensors of the subsystem. All this information is provided in the PDL file, which is loaded when the agent is instantiated (see Listing 5.1). Therefore, each PDL contain only the components and relations of the immediate neighbors, with normally different PDL for each Supervisor Agent as the neighborhood of each agent is different.

With *FAST*, the analysis of the system and its partition into subsystems is performed at design time. Thus, this information is used to define the system architecture and deploy the OPC servers according to the subsystem partitions. Each Supervisor Agent requires local access through its associated OPC Server to the measured variables of the associated components.

The Supervisor Agents run in a Java Virtual Machine, providing almost total independence of the target computer operating system and hardware. The agent names in the PDL provide the server address embedded in the name (see Listing 5.1) as the right part after the '@' symbol. This is a standard agent naming convention and allows the JADEX platform to connect to remote agents residing in other Java Virtual Machines connected in a network. This architecture allows a large number of possible configurations, with some agents running in the same Java Virtual Machine, some agents running in the same computer but in separated Java Virtual Machines (or Virtual Servers) or agents running remotely in several computers connected through Internet.

5.3 Agent Request of an External Residual Calculation

To calculate the residuals in a distributed configuration, the Supervisor Agent will implement the Remote_Calculate_Relation_Plan which will be activated when a request from other agent is received to calculate a relation involving local components and provide the resulting value to the message originator agent. The receiver will be able to complete the calculation of the fault vector and compare it with the local portion of the Fault Signature Matrix. However, it should be taken into account, that in the most

LISTING 5.2: Components associated to the Supervision Agent in the PDL

```
<Components>
<Component Name="Tank1" Tag_Error="TK_1001_01_ERR"
agent="SA1" />
<Component Name="Tank2" Tag_Error="TK_1002_01_ERR"
agent="SA1"/>
<Component Name="Tank3" Tag_Error="TK_1003_01_ERR"
agent="SA2"/>
<Component Name="Tank4" Tag_Error="TK_1004_01_ERR"
agent="SA2"/>
...
</Components>
```

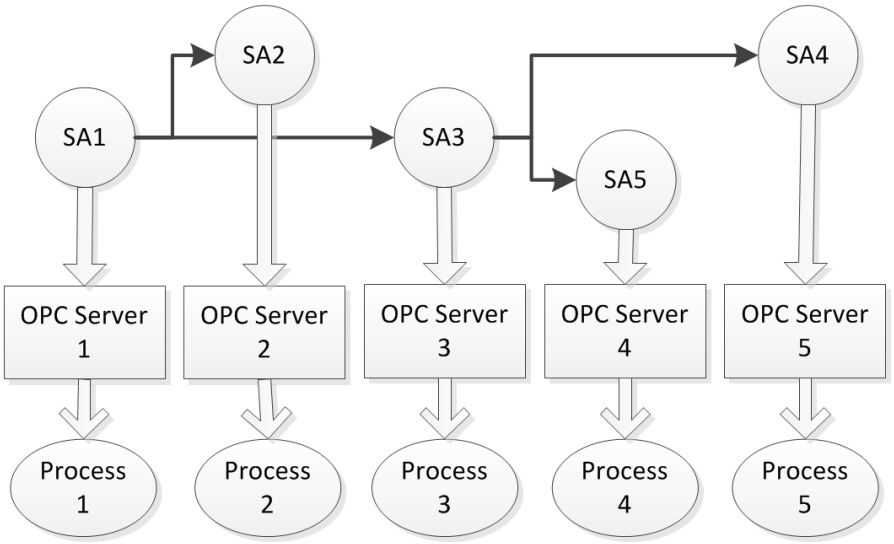


FIGURE 5.2: Complex distributed FDI.

complex case, the second Supervisor Agent might require access to another agent to calculate the relation, depending on the complexity of the system (see Figure 5.2) .

5.4 Time Constraints

In a distributed configuration, a special attention must be given to the communication delays. A misalignment of the values obtained from different Supervision Agents distributed in far locations can cause false alarms just because old and new samples are mixed and cause the residual to evaluate different from 0 [Blesa et al., 2014].

Initially, the proposed architecture is feasible in soft-real time processes. Normally, the set OPC Server and SCADA has a sample time in the order of the second. Just by this restriction the level of concern about communication delays is minor. Clearly, the processes supervised should have a cycle time much higher. On the other hand, the computation cycle of the residual evaluation by the Supervisor Agent is also defined in the order of a second.

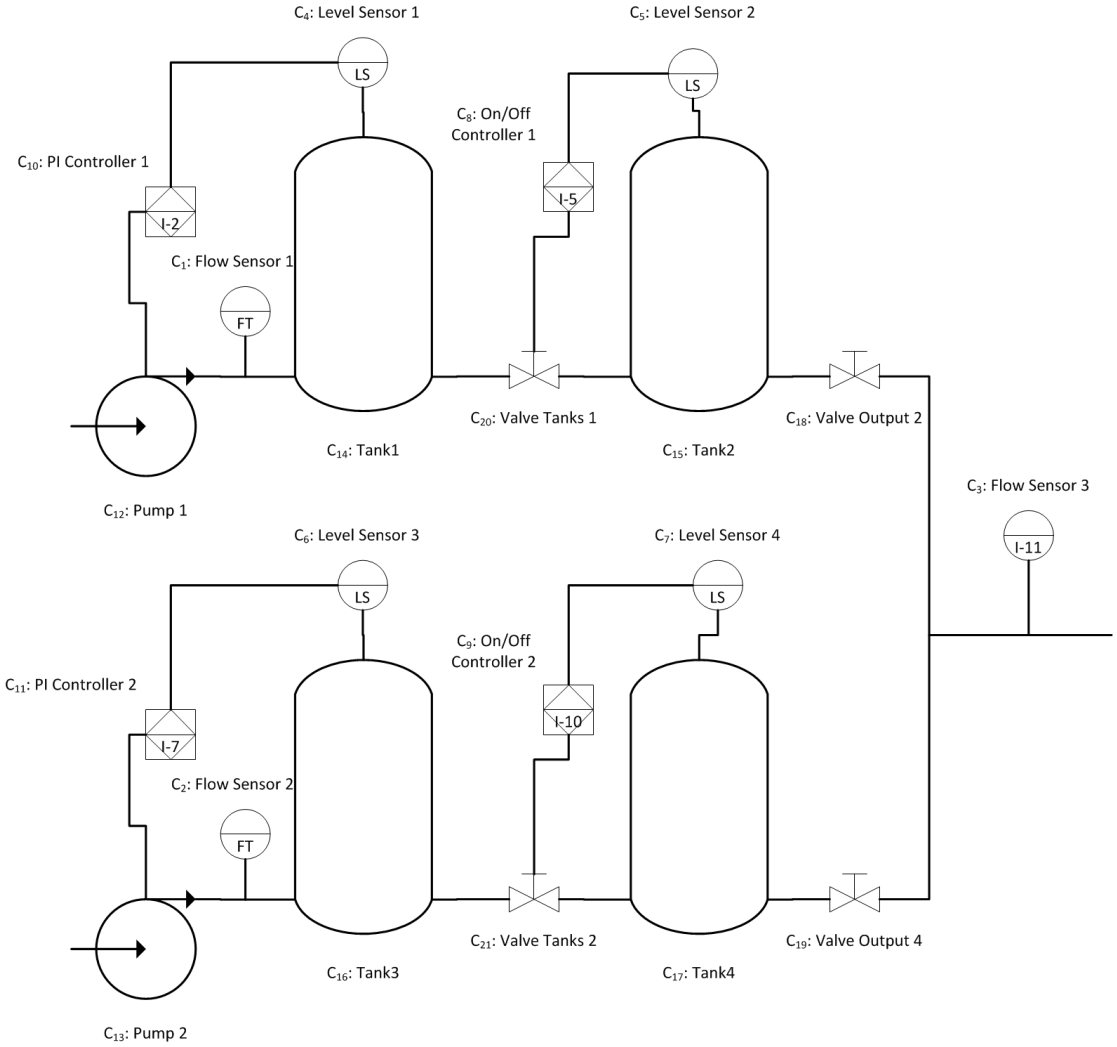


FIGURE 5.3: Water distribution example.

Therefore, in order to be able to detect time misalignments between among consolidated diagnosis data from several Supervisor Agents, all information exchanged shall provide the time signature and all nodes should be synchronized periodically to avoid time deviations over the sampling period. Although this is a transversal problem and well identified in distributed systems, it has implementation implications in the format of the messages exchanged between Supervision Agents and the requirement of considering the synchronization between nodes when deploying the architecture.

5.5 A Water Distribution Example

Consider the system presented in Figure 5.3. After processing the corresponding PDL with the *FAST* Analysis Tool and performing the structural analysis we can identify 32 different relations, 34 process variables and 21 components. The resulting fault signature matrix $\mathbf{F}$ is indicated in Figure 5.4. In $\mathbf{F}$, we can identify that there are 11 residuals $\mathcal{Z} = \{z_1, ..., z_{11}\}$. After applying the partitioning algorithm based on the components coupling, the resulting fault signature matrix is indicated in the Figure 5.5. In this matrix, we can identify two independent groups of components which are candidate to

be subsystems: C_a ={$c_1, c_{10}, c_{12}, c_{14}, c_4$} and C_b = {$c_{11}, c_{13}, c_{16}, c_2, c_6, c_9, c_7, c_{21}, c_{17}, c_{19}$}. In addition, we can identify a group of shared components C_s ={$c_8, c_5, c_{20}, c_{15}, c_{18}, c_3$}. The shared components set C_s correspond to the group of sensors, controller and valve belonging to the Tank 2 and the Flow Sensor 3 placed at the output of the system. This result is logic, since the partitioning in this case is quite trivial just looking at the system diagram. The variable which relates the two subsystems is the output flow which is measured by the Flow Sensor 3. Therefore, just by knowing the flow from one of the branches through the Flow Sensor 3, the flow of the other branch can be determined.

To assign the shared components, *FAST* calculates the coupling level but now taking into account the subsystem candidates. That is, by defining two residual subsets $\mathcal{Z}_a = \{z_1, z_4, z_2, z_5, z_3\}$ and $\mathcal{Z}_b = \{z_{11}, z_{10}, z_9, z_7, z_6, z_8\}$, we can identify which components are with a higher coupling with respect to each residuals subset, as indicated in *Definition 2*:

$$\zeta(c_8, \mathcal{Z}_a) = 0.42 \text{ and } \zeta(c_8, \mathcal{Z}_b) = 0.32$$
$$\zeta(c_3, \mathcal{Z}_a) = 0.36 \text{ and } \zeta(c_3, \mathcal{Z}_b) = 0.48$$

resulting that components $c_8, c_5, c_{20}, c_{15}, c_{18}$ have a higher coupling level with respect to C_a and c_3 has a higher coupling level C_b. With these results, we can finally assign the components to the two subsystems identified:

- C'_a ={$c_1, c_{10}, c_{12}, c_{14}, c_4, c_8, c_5, c_{20}, c_{15}, c_{18}$}
- C'_b = {$c_{11}, c_{13}, c_{16}, c_2, c_6, c_9, c_7, c_{21}, c_{17}, c_{19}, c_3$}

This information is fed into the PDL by defining the subsystems as Supervisor Agents and identifying for each component to which Supervisor Agent is assigned (see Listing 5.1 and Listing 5.2).

Therefore for this example, two Supervisor Agents will be instantiated: the SA_1 will identify as local the components set C'_a and the SA_2 will identify as local the components set C'_b. For this case, there will be only one interface between them, although in more complex cases there might be multiple supervisor agents with multiple interfaces. In this case, the PDL will be the same for each agent as they contain the components and relations of the immediate neighbors.

Each agent executes periodically the plan Calculate_Residuals_Plan which evaluates all residuals to which the local components are detectable, note that the other residuals do not provide useful information, as they do not belong to the supervised subsystem. In the example, the SA_1 calculates the residuals $\mathcal{Z}_a = \{z_1, z_4, z_2, z_5, z_3\}$ plus z_{11}. The residuals from $\mathcal{Z}_a$ can be solved locally. However, the residual z_{11} requires to calculate some relations which are remote ($r_{32}, r_{20}, r_{26}, r_{17}$). These relations are requested to be solved by the SA_2.

At the same time, the Supervisor Agent SA_2 will be as well executing the same plan. For the SA_2, the residuals which can be solved locally are z_9, z_7, z_6 and z_8 while z_{10} and z_{11} require that some relations are solved by the Supervisor Agent SA_1.

In this application example, the two agents SA_1 and SA_2 run in the same Java Virtual Machine and access to the same OPC server connected to the Proficy iFix SCADA. The values are provided by the *FAST* Simulator which communicates as well with the OPC server but providing the values that should be obtained from the physical process. With this configuration, it is very easy to test the diagnosibility of the system and modify its configuration to increase it. In this case, by adding the residuals z_{19} and

Res	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20	c21
z1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0
z2	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
z3	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
z4	1	0	0	1	1	0	0	1	0	1	0	1	0	1	1	0	0	1	0	1	0
z5	0	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0	1	0
z6	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0
z7	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
z8	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1
z9	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	1	1	0	1	0	1
z10	0	0	1	0	0	1	1	0	1	0	1	0	0	0	0	1	1	0	1	0	1
z11	0	0	1	0	1	0	1	1	1	0	0	0	0	0	1	0	1	1	1	1	1

FIGURE 5.4: Resulting Fault Signature Matrix **F**.

Res	c1	c10	c12	c14	c4	c8	c5	c20	c15	c18	c3	c11	c13	c16	c2	c6	c9	c7	c21	c17	c19
z1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
z4	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
z2	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
z5	0	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
z3	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Z11	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1
Z10	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	1	1	1	1
Z9	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
z7	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0
z6	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
z8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

FIGURE 5.5: Fault signature matrix after applying the partitioning algorithm.

z_{11} to the Fault Signature Matrix of the Subsystem SA_1, we increase the capability of identifying the faulty component. For instance, the Hamming distance between the fault signature corresponding to fc_{18} and fc_3 without the additional residuals is 1 and with the additional residuals is 2. The addition of these residuals, however, forces the supervisor agent SA_1 to request the value for some of the relations to the SA_2.

Chapter 6

Conclusions and Recommendations

6.1 Conclusions

Through the development of this work, it has been possible to demonstrate that the Structural Analysis is a valuable technique to obtain fault diagnosis information, which can be used in real-time to perform on-line fault diagnosis. The utilization of this technique, facilitated by a tool which automates the analysis and implementation process, approaches its systematic application to the industry. The FDI analysis should be established as a mandatory step in the design of automated industrial processes providing feedback to the design, components and topology with the final objective of increasing the process reliability and safety. On the other hand, the deployment of FDI integrated in the SCADA supervision system and using as a support the software agents architecture, it has been also proven as a feasible and a worthy way of implementing distributed fault diagnosis. The proposed *FAST* tool provides an easy path from the design phase to the implementation phase by using the same source of information. The tool used connected to the process, provides the possibility of feeding the FDI information directly to the SCADA supervision software, where the fault diagnosis results can be presented directly in the synoptic together with the rest of the supervision information. Furthermore, the analysis presented can be extended to other application environments beyond the classic industrial process. The simulation part could be directly reused for any complex system, just by incorporating the component models in *FAST*, and the same concept of feeding the real-time diagnosis with the information utilized in the design phase could be also reused by adapting the residual evaluation algorithms to any target environment.

6.2 Future Work

6.2.1 FAST in Switched Affine Systems

Almost all systems mix discrete-event and continuous modes of operation behaving as hybrid systems. The design of control algorithms is progressively incorporating the possibility of mode changes due to discrete-events which cause to reconfigure the system and requires the adaptation of the control system to the new mode. The way these discrete events and mode changes are incorporated into the Structural Analysis and their effects on the residuals calculation should be evaluated. It should be also evaluated the possibility that *FAST* implements the functionality to manage these discrete-events mode changes incorporating the changes into the models used for the residuals calculation and therefore generalize the fault diagnosis capability beyond only one mode of operation.

6.2.2 FAST Extension to Hybrid Systems

As commented in the introduction part of this thesis, in the space sector almost all systems are critical. Beyond the implementation during the design phase of the FMEA analysis and incorporating hardware redundancy to avoid that any single failure causes a mission loss, there are a lot of possibilities to incorporate analytical redundancy and the proper Structural Analysis to provide in real-time information about faults. Even though the detailed analysis and all hardware redundancy, fault detection and isolation continue being a problem for satellite systems. The faults are detected late and it is easy that there is not enough information to identify uniquely the faulty component. For instance, if there is an over-temperature detection by a sensor in one of the spacecraft subsystems, the autonomous reaction will be to switch-off this system which can be the scientific payload which carries out the major function in terms of science result in the spacecraft. However, it is possible that the faulty component is not the payload but the temperature sensor chain. The structural analysis proposed by *FAST* and the modeling of the spacecraft components at low level can prevent these problems by identifying where sensor redundancy is required and using the analytical redundancy improve the fault identification, eventually improving the science return. The application of *FAST* to this projects in the implementation phase would be straight forward by using the *FAST* simulation capabilities. However, the deployment in a real system would require to move the implementation of the on-line supervision to a real-time embedded environment, with lack of the possibility of using a software agents framework. This is a considerable task in terms of time and effort but affordable and in fact the resulting software could be reused among several projects.

6.2.3 FAST in MELiSSA

The deployment of *FAST* in the MELiSSA Pilot Plant is an interesting exercise, more if the fact that it was the inspiring project for the development of this work is considered. However, the major interest to incorporate *FAST* to the supervision system would be to include the process model as an additional component as part of the FDI analysis. That is, apart from considering for the FDI analysis the ancillary components (pumps, valves, sensors, etc.) the biological process model for the MELiSSA bio-reactors could be also incorporated providing additional fault diagnosis information. For instance, the compartment CIVa is a photo-trophic compartment, which means that the algae culture in the bio-reactor needs light energy to perform the photosynthesis. The metabolic model of the algae, which is the *Spirulina Platensis*, is known and therefore the model of growth rate of the biomass, related to the light energy and the nutrients compounds concentration, can be calculated. As the biomass concentration is also measured it would be possible by adding this process information to the fault diagnosis model to detect faults also related with the biological process. On the other hand, the MELiSSA Pilot Plant is a real distributed process, as it has different bio-reactors interconnected. The system would benefit of the capability of *FAST* to implement distributed diagnosis.

Appendix A

FAST User Manual

A.1 The Process Definition File

The definition of the Process and Instrumentation Diagram (P&ID) is a common step in industrial processes design. In fact, the diagram is the result of the design, where the topology, the components and the control are indicated. The diagram will be used by the process engineers to implement the actual process. It is from this diagram that *FAST* can obtain the information to start the FDI analysis generating the Process Definition File (PDL).

A.1.1 Generation of the PDL

Any process can be represented with a PDL. The only pre-condition is that the models of the components are available. In case that a model has to be incorporated to *FAST* see Appendix B.

As defined in Section 3.1, the PDL is composed of:

- Process Variables
- Components
- Relations

In order to describe the process of creating a PDL the example of the two-tanks system will be used. A simple representation of the P&ID is indicated in Figure A.1.

The PDL can be generated with any plain text processor or XML editor. To start the generation of the PDL we need to identify the components which participate in the

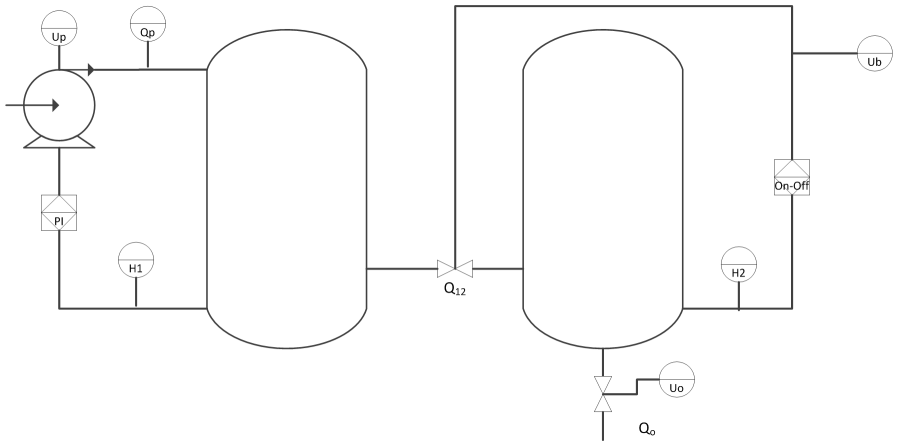


FIGURE A.1: Two-tanks process example

process. For the example the list of components with the relations corresponding to the component models is indicated in the listing A.1.

1. Pump Considering the most simple case as a linear relation between the output flow and the control signal:
$$Q = a \cdot U_c + b \tag{A.1}$$
$$U_c = U_{mc} \tag{A.2}$$
2. Tank Defining Q_i for input flow, Q_o for output flow, and H level:
$$\frac{dH}{dt} = \frac{1}{A}\left(Q_i(t) - Q_o(t)\right) \tag{A.3}$$
$$\dot{H} = \frac{dH}{dt} \tag{A.4}$$
3. Flow sensorç For flow Q, without considering error nor adaptation to the measure range of the sensor:
$$Q_m = Q \tag{A.5}$$
4. Level sensor Without considering error nor adaptation to the measure range of the sensor:
$$H_m = H \tag{A.6}$$
5. Valve Being C_v the valve hydraulic coefficient:
$$Q_o = C_v \cdot \sqrt{H} \tag{A.7}$$
6. PI controller
$$U_c = K_{pi}(H_c - h_m(t)) + K_{ii}\int (H_c - h_m(t))dt \tag{A.8}$$
7. On-off controller.
$$U_b = \begin{cases} 1 & \text{if } x \geq P \\ 0 & \text{otherwise} \end{cases} \tag{A.9}$$
with P as input parameter.

The corresponding representation in the PDL is indicated in the listing A.1:

LISTING A.1: PDL of the two-tanks example

```
<?xml version="1.0" encoding="UTF-8"?>
<Process>
<ProcessVars>
        <ProcessVar Name="Flow Input" Symbol="Qp" Init="0" Units="m3/s" Tag="FTE_1001_01"/>
        <ProcessVar Name="Flow Input Measured" Symbol="mQp" Units="m3/s" Tag="FT_1001_01"/>
        <ProcessVar Name="Tank1 Level" Symbol="H1" Units="m" Init="0.2" Tag="LSE_1001_01"/>
        <ProcessVar Name="Tank2 Level" Symbol="H2" Units="m" Init="0.2" Tag="LSE_1002_01"/>
        <ProcessVar Name="Tank1 Level Measured" Symbol="mH1" Units="m" Tag="LS_1001_01"/>
        <ProcessVar Name="Tank2 Level Measured" Symbol="mH2" Units="m" Tag="LS_1002_01"/>
        <ProcessVar Name="Flow Tank1 to Tank2" Symbol="Q12" Units="m3/s" Init="0" Tag="FTE_1001_02"/>
        <ProcessVar Name="Flow Tank2 out" Symbol="Qo" Units="m3/s" Init="0" Tag="FTE_1002_01"/>
        <ProcessVar Name="Tank1 Level Delta" Symbol="dH1" Units="m" Init="0"/>
        <ProcessVar Name="Tank2 Level Delta" Symbol="dH2" Units="m" Init="0"/>
        <ProcessVar Name="Control Signal" Symbol="Up" Units="" Init="0"/>
        <ProcessVar Name="Control Signal Measured" Symbol="mUp" Init="0" Units="" Tag="PP_1001_01"/>
        <ProcessVar Name="Valve Tank1 to Tank2" Symbol="Ub" Units="" Init="1"/>
        <ProcessVar Name="Valve Tank1 to Tank2 Measured" Symbol="mUb" Units="" Init="0" Tag="VT_1001_01"/>
        <ProcessVar Name="Valve Tank2 output" Symbol="Uo" Units="" Init="0"/>
        <ProcessVar Name="Valve Tank2 output Measured" Symbol="mUo" Units="" Init="0" Tag="VT_1002_01"/>
</ProcessVars>
```

```
<Parameters>
        <Parameter Name="SampleTime" Value="1.0"/>
</Parameters>
<Components>
        <Component Name="Tank1" Tag_Error="TK_1001_01_ERR"/>
        <Component Name="Tank2" Tag_Error="TK_1002_01_ERR"/>
        <Component Name="Pump" Tag_Error="PP_1001_01_ERR"/>
        <Component Name="PI␣Controller" Tag_Error="PI_1001_01_ERR"/>
        <Component Name="Valve␣Tanks" Tag_Error="VT_1001_01_ERR"/>
        <Component Name="Valve␣Output" Tag_Error="VT_1002_01_ERR"/>
        <Component Name="Level␣Sensor1" Tag_Error="LS_1001_01_ERR"/>
        <Component Name="Level␣Sensor2" Tag_Error="LS_1002_01_ERR"/>
        <Component Name="Flow␣Sensor1" Tag_Error="FT_1001_01_ERR"/>
        <Component Name="On/Off␣Controller" Tag_Error="CT_1002_01_ERR"/>
</Components>
<Relations>
        <Relation Name="R01" Type="Sensor">
                <Component>Level Sensor1</Component>
                <ProcessVar>H1</ProcessVar>
                <ProcessVarMeasured>mH1</ProcessVarMeasured>
                <Precision>0.001</Precision>
        </Relation>
        <Relation Name="R02" Type="Sensor">
                <Component>Level Sensor2</Component>
                <ProcessVar>H2</ProcessVar>
                <ProcessVarMeasured>mH2</ProcessVarMeasured>
                <Precision>0.001</Precision>
        </Relation>
        <Relation Name="R03" Type="Sensor">
                <Component>Flow Sensor1</Component>
                <ProcessVar>Qp</ProcessVar>
                <ProcessVarMeasured>mQp</ProcessVarMeasured>
                <Precision>0.000001</Precision>
        </Relation>
        <Relation Name="R04" Type="Sensor">
                <Component>Valve Tanks</Component>
                <ProcessVar>Ub</ProcessVar>
                <ProcessVarMeasured>mUb</ProcessVarMeasured>
        </Relation>
        <Relation Name="R05" Type="Sensor">
                <Component>Valve Output</Component>
                <ProcessVar>Uo</ProcessVar>
                <ProcessVarMeasured>mUo</ProcessVarMeasured>
        </Relation>
        <Relation Name="R06" Type="Sensor">
                <Component>Pump</Component>
                <ProcessVar>Up</ProcessVar>
                <ProcessVarMeasured>mUp</ProcessVarMeasured>
        </Relation>
        <Relation Name="R07" Type="Pump">
                <Component>Pump</Component>
                <ProcessVarFlow>Qp</ProcessVarFlow>
                <ProcessVarControl>Up</ProcessVarControl>
                <Gain>1</Gain>
                <Offset>0</Offset>
        </Relation>
        <Relation Name="R08" Type="ControllerPID">
                <Component>PI Controller</Component>
                <ParamKi>5e-6</ParamKi>
                <ParamKp>1e-3</ParamKp>
                <ProcessVarControl>Up</ProcessVarControl>
                <ProcessVarMeasured>mH1</ProcessVarMeasured>
                <SetPoint>0.8</SetPoint>
        </Relation>
        <Relation Name="R09" Type="ControllerOnOff">
                <Component>On/Off Controller</Component>
                <ProcessVarControl>Ub</ProcessVarControl>
                <ProcessVarMeasured>mH2</ProcessVarMeasured>
                <HighLimit>0.6</HighLimit>
                <LowLimit>0.1</LowLimit>
        </Relation>
        <Relation Name="R10" Type="Valve2Levels">
                <Component>Valve Tanks</Component>
                <ProcessVarFlow>Q12</ProcessVarFlow>
                <ProcessVarLevel1>H1</ProcessVarLevel1>
                <ProcessVarLevel2>H2</ProcessVarLevel2>
                <ProcessVarControl>Ub</ProcessVarControl>
                <Coefficient>1.596E-4</Coefficient>
        </Relation>
        <Relation Name="R11" Type="ValveLevel">
                <Component>Valve Output</Component>
                <ProcessVarFlow>Qo</ProcessVarFlow>
                <ProcessVarLevel>H2</ProcessVarLevel>
                <ProcessVarControl>Uo</ProcessVarControl>
                <Coefficient>1.596E-4</Coefficient>
        </Relation>
        <Relation Name="R12" Type="Derivative">
                <Component>Tank1</Component>
                <ProcessVar>H1</ProcessVar>
                <ProcessVarDerivative>dH1</ProcessVarDerivative>
        </Relation>
        <Relation Name="R13" Type="Derivative">
                <Component>Tank2</Component>
                <ProcessVar>H2</ProcessVar>
                <ProcessVarDerivative>dH2</ProcessVarDerivative>
        </Relation>
```

```
117         <Relation Name="R14" Type="Tank">
118                 <Component>Tank1</Component>
119                 <ProcessVarDeltaLevel>dH1</ProcessVarDeltaLevel>
120                 <ProcessVarFlowIn>Qp</ProcessVarFlowIn>
121                 <ProcessVarFlowOut>Q12</ProcessVarFlowOut>
122                 <TankArea>0.0154</TankArea>
123         </Relation>
124         <Relation Name="R15" Type="Tank">
125                 <Component>Tank2</Component>
126                 <ProcessVarDeltaLevel>dH2</ProcessVarDeltaLevel>
127                 <ProcessVarFlowIn>Q12</ProcessVarFlowIn>
128                 <ProcessVarFlowOut>Qo</ProcessVarFlowOut>
129                 <TankArea>0.0154</TankArea>
130         </Relation>
131 </Relations>
132 </Process>
```

A.1.2 General Conventions

To generate the PDL some general conventions will help to better understand the results and diagnose possible problems when parsing the file.

The components should have meaningful names, possibly related with their reference in the P&ID diagram.

The naming of the process variables should be consistent, use the same name in each relation where each process variable is involved. Process variables define the links between relations, that is, two relations with the same process variable are connected. *FAST* uses these connections to perform the structural analysis. In the examples the following names are used:

- Q: Flow rate
- H: Tank level

The measured process variables name starts with "m". For example "Q" for the not measured variable and "mQ" for the measured process variable.

If there is more than one process variable of the same physical property, that is, more than one tank level or liquid flow, add a suffix number to the process variable: H_1 or Q_2.

The Relations should be named with a sequential number "R01, R02, R03, ...".

A.2 FAST Models

The version 1.0 of *FAST* incorporate a simple version of several component models used to implement the case studies indicated in Sections 4.4 and 5.5. Although the models can be much more complex, with this simple version it is possible to evaluate the feasibility of the tool and the possibilities of its utilization. The models are detailed in the subsections below:

A.2.1 Controller On-Off

The model of the On-off controller is indicated below:

$$U_b = \begin{cases} 1 & \text{if } x < L_{high}, \\ 1 & \text{if } x \geq L_{low}, \\ 0 & \text{otherwise.} \end{cases} \tag{A.10}$$

The representation in the PDL is as follows:

```
<Relation Name="[name of the relation]" Type="ControllerOnOff">
     <Component>On/Off Controller</Component>
     <ProcessVarControl>[From process vars]</ProcessVarControl>
     <ProcessVarMeasured>[From process vars]</ProcessVarMeasured>
     <HighLimit>[High limit value]</HighLimit>
     <LowLimit>[Low limit value]</LowLimit>
</Relation>
```

The description of the attributes is the following:

- Relation Name = Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type = Shall be "ControllerOnOff". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarControl: Controlled process variable. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarMeasured: Measured process variable. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- HighLimit: Output transition upper limit value.
- LowLimit: Output transition lower limit value.

A.2.2 Controller PID

The model of a Process,Integrative and Derivative controller is indicated below:

$$U_c = K_p(x_c - x_m(t)) + K_i \int (x_c - x_m(t))dt + K_d * \frac{d(x_c - x_m(t))}{dt} \tag{A.11}$$

The discrete representation of the controller is implemented in *FAST* as follows, if we represent the error as $\varepsilon(k) = x_c - x_(k)$, being x_c the set-point and $x_(k)$ the model variable corresponding to the measured value, using the trapezoidal rule to calculate the integrative part and the differences approximation for the derivative part, considering the sample time $T = 1$:

$$U_c(k) = K_p * \varepsilon(k) + K_i * \varepsilon(k) * T + K_d * [\varepsilon(k) - \varepsilon(k-1)]/T + U_c(k-1) \tag{A.12}$$

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation]" Type="ControllerPID">
     <Component>[Name of the component]</Component>
     <ParamKi>[Value of the Ki parameter]</ParamKi>
     <ParamKd>[Value of the Kd parameter]</ParamKi>
     <ParamKp>[Value of the Kp parameter]</ParamKp>
     <ProcessVarControl>
          [Controlled process variable]
     </ProcessVarControl>
     <ProcessVarMeasured>
```

```
          [Measured process variable]
     </ProcessVarMeasured>
     <SetPoint>[Set point value]</SetPoint>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "ControllerPID". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ParamKi: Value of the integration part gain (notation 1e-6 is possible).
- ParamKd: Value of the derivative part gain (notation 1e-3 is possible).
- ParamKp: Value of the proportional part gain.
- ProcessVarControl: Controlled process variable. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarMeasured: Measured process variable. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- SetPoint: Set point of the measured variable objective of the controller.

A.2.3 Derivative

The model of a derivative relation is implemented as follows:

$$y(t) = \frac{dx}{dt} \tag{A.13}$$

The discrete form is calculated using the differences approximation:

$$y(k) = \frac{x(k) - x(k-1)}{T} \tag{A.14}$$

being T the sampling period.
The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation]" Type="Derivative">
     <Component>[Name of the component]</Component>
     <ProcessVar>[Name of the process variable]</ProcessVar>
     <ProcessVarDerivative>
          [Name of the derivative process variable]
     </ProcessVarDerivative>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "Derivative". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVar: Process variable which will be derived. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarDerivative: Process variable corresponding to the derivative of the ProcessVar. Provide the name of the process variable as indicated in the list of process variables of the PDL.

A.2.4 Flow Union

The model the flow output a flow union relation is implemented as follows:

$$Q_o(t) = Q_a(t) + Q_b(t) \tag{A.15}$$

being the Q_o the output flow resulting of the sum of the flows Q_a and Q_b.
The representation in the PDL is as follows:

```
<Relation Name=" [Name of the relation]" Type="Flow Union">
     <Component>[Name of the compoenent]</Component>
     <ProcessVarFlowIn1>
          [Process variable flow input a]
     </ProcessVarFlowIn1>
     <ProcessVarFlowIn2>
          [Process variable flow input b]
     </ProcessVarFlowIn2>
     <ProcessVarFlowOut>
          [Process variable flow output]
     </ProcessVarFlowOut>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "Flow Union". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarFlowIn1: Process variable which will corresponds to the flow input a. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarFlowIn2: Process variable which corresponds to the flow input b. Provide the name of the process variable as indicated in the list of process variables of the PDL.

- ProcessVarFlowOut: Process variable which corresponds to the output flow. Provide the name of the process variable as indicated in the list of process variables of the PDL.

A.2.5 Pump

The model of the flow output of a pump is implemented as follows:

$$Q(t) = a \cdot U_c(t) + b \tag{A.16}$$

considering the most simple case as a linear relation between the output flow $Q(t)$ and the control signal $U_c(t)$.

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation]" Type="Pump">
     <Component>[Name of the component]</Component>
     <ProcessVarFlow>[Process variable flow output]</ProcessVarFlow>
     <ProcessVarControl>
          [Process variable control signal]
     </ProcessVarControl>
     <Gain>[Value corresponding to the gain]</Gain>
     <Offset>[Value corresponding to the offset]</Offset>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "Pump". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarFlow: Process variable which will corresponds to the flow output provided by the pump. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarControl: Process variable which corresponds to control signal provided by the controller. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- Gain: Proportional gain for the linearization of the pump action.
- Offset: Offset for the linearization of the pump action.

A.2.6 Sensor

The model of a sensor is implemented as follows:

$$x_m(t) = \begin{cases} x_{low} & \text{if } x < L_{low}, \\ x_{high} & \text{if } x > L_{high}, \\ x(t) + \sigma(t) & \text{if } L_{low} \leq x(t) \leq L_{high} \end{cases} \tag{A.17}$$

considering the adaptation to the measure range of the sensor x_{high}, x_{low} and taking into account the sensor precision as $\sigma(t)$, providing an added white noise signal simulating the sensor error due to its precision.

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation]" Type="Sensor">
     <Component>[Name of the component]</Component>
     <ProcessVar>[Name of the process variable]</ProcessVar>
     <ProcessVarMeasured>
          [Name of the measured process variable]
     </ProcessVarMeasured>
     <UpperLimit>[Value of the upper limit]</UpperLimit>
     <LowerLimit>[Value of the lower limit]</LowerLimit>
     <Precision>[Value of the sensor precision]</Precision>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "Pump". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVar: Process variable which is measured. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarMeasured: Process variable results from the measurement. Provide the name of the process variable as indicated in the list of process
-
- UpperLimit: Value of the saturation upper limit of the sensor.
- LowerLimit: Value of the saturation lower limit of the sensor.
- Precision: Value of the precision of the sensor.

A.2.7 Tank

The model of the level variation in a cylindrical tank is implemented as follows:

$$\frac{dH}{dt} = \frac{1}{A}\left(Q_i(t) - Q_o(t)\right) \tag{A.18}$$

being dH the derivative of the tank level, A the tank section in meters, and Q_i the flow input and Q_o the flow output. The discrete representation uses the differences approximation for the derivative part $\Delta h(k) = (h(k) - h(k-1))/T$.

The residual is then calculated as follows considering the sampling period $T = 1$:

$$\Delta h(k) = \frac{1}{A}\left(Q_{mi}(k) - Q_{mo}(k)\right) \tag{A.19}$$

with Q_{mi} the measured flow input and Q_{mo} the measured flow output if sensors are available or from other relations taking only measured variables to calculate it.

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation]" Type="Tank">
     <Component>[Name of the component]</Component>
     <ProcessVarDeltaLevel>
          [Process variable of the level derivative]
     </ProcessVarDeltaLevel>
     <ProcessVarFlowIn>
          [Process variable flow input]
     </ProcessVarFlowIn>
     <ProcessVarFlowOut>
          [Process variable flow ouptut]
     </ProcessVarFlowOut>
     <TankArea>[Value of the tank area section]</TankArea>
</Relation>
```

The description of the attributes is the following:

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "Tank". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarDeltaLevel: Process variable corresponding to the level derivative in meters. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarFlowIn: Process variable corresponding to the flow input. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarFlowOut: Process variable corresponding to the flow output. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- TankArea: Value of the tank area section in meters.

A.2.8 Valve Level

The model of the flow output of a valve connected to a tank output is implemented as follows:

$$Q_o(t) = C_v \cdot \sqrt{H(t)} \tag{A.20}$$

being C_v the valve hydraulic coefficient, $Q_o(t)$ the flow output of the valve and $H(t)$ the level of the tank.

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation" Type="ValveLevel">
     <Component>[Name of the component]</Component>
     <ProcessVarFlow>[Process variable flow]</ProcessVarFlow>
     <ProcessVarLevel>[Process variable level]</ProcessVarLevel>
     <ProcessVarControl>[Process variable control]</ProcessVarControl>
     <Coefficient>[Value of the valve coefficient]</Coefficient>
</Relation>
```

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.
- Relation Type: Shall be "ValveLevel". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarFlow: Process variable corresponding to the flow output of the valve in m^3/s. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarLevel: Process variable corresponding to the level of the tank in meters to which the valve is connected. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarControl: Process variable corresponding to the valve which controls the valve status (open = 1/ closed = 0). Provide the name of the process variable as indicated in the list of process variables of the PDL.
- Coefficient: Value of the coefficient of the valve.

A.2.9 Valve Two-Levels

The model of the output flow of a valve connected to a tank output and to a tank input is implemented as follows:

$$Q_{12}(t) = C_v \cdot sgn(h_1 - h_2)\sqrt{|h_1 - h_2|} \cdot U_c \tag{A.21}$$

being C_v the valve hydraulic coefficient, $Q_{12}(t)$ the flow output of the valve and h_1 the level of the tank connected to the valve input and h_2 the level of the tank output in meters.

The representation in the PDL is as follows:

```
<Relation Name="[Name of the relation" Type="ValveLevel">
     <Component>[Name of the component]</Component>
     <ProcessVarFlow>[Process variable flow]</ProcessVarFlow>
     <ProcessVarLevel1>[Process variable level1]</ProcessVarLevel>
     <ProcessVarLevel2>[Process variable level2]</ProcessVarLevel>
     <ProcessVarControl>[Process variable control]</ProcessVarControl>
     <Coefficient>[Value of the valve coefficient]</Coefficient>
</Relation>
```

- Relation Name: Name of the relation, for example "R01". Use it to identify the relation in the structural analysis matrices.

- Relation Type: Shall be "ValveLevel". Defines the type of the relation.
- Component: Component to which the relation belongs. Provide the name of the component as indicated in the components list in the PDL.
- ProcessVarFlow: Process variable corresponding to the flow output of the valve in m^3/s. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarLevel1: Process variable corresponding to the level of the tank in meters to which the valve input is connected. Provide the name of the process variable as indicated in the list of process variables of the PDL.
- ProcessVarLevel2: Process variable corresponding to the level of the tank in meters to which the valve output is connected. Provide the name of the process variable as indicated in the list of process variables of the PDL
- ProcessVarControl: Process variable corresponding to the valve which controls the valve status (open = 1/ closed = 0). Provide the name of the process variable as indicated in the list of process variables of the PDL.
- Coefficient: Value of the coefficient of the valve.

A.3 FAST Simulator

As indicated in Chapter 3 *FAST* is composed of two main applications. The *FAST Simulator* and the *Supervisor Agent*. The *FAST Simulator* is a stand-alone Java application (JAR file) which can be started from the command line. As a Java application is fully platform independent and only the Java run-time engine (JRE) corresponding to the target operating system is required. The exercises presented in this work have been mainly executed in a Microsoft Windows 7 environment. The CPU, memory and hard disk space requirements are not very relevant as the Java application is light-weight and will run in mostly in all modern hardware configurations.

A.3.1 Loading the PDL

Once the application is started, from the menu 'Archive/Open process...' it is possible to load a PDL file. Just after loading a PDL file following processes are run sequentially:

- PDL Parsing: The PDL file is parsed and component models are instantiated.
- Structural Matrix analysis: The application generates the *Structural Matrix*. The *Structural Matrix* is generated in a 'comma-separated-values' (csv) file named "struct_matrix.csv".
- Perfect Matching analysis: The application generates the *Perfect Matching* matrix. The *Perfect Matching* is generated in a 'comma-separated-values' (csv) file named "perfect_matching.csv". The primary relations are indicated with an asterisc '*' next to the "1" indicating the association between the relation and the process variable.
- Fault Signature Matrix analysis: The application generates the *Fault Signature Matrix*. The *Fault Signature Matrix* is generated in a 'comma-separated-values' (csv) file named "fault_signature_matrix.csv".

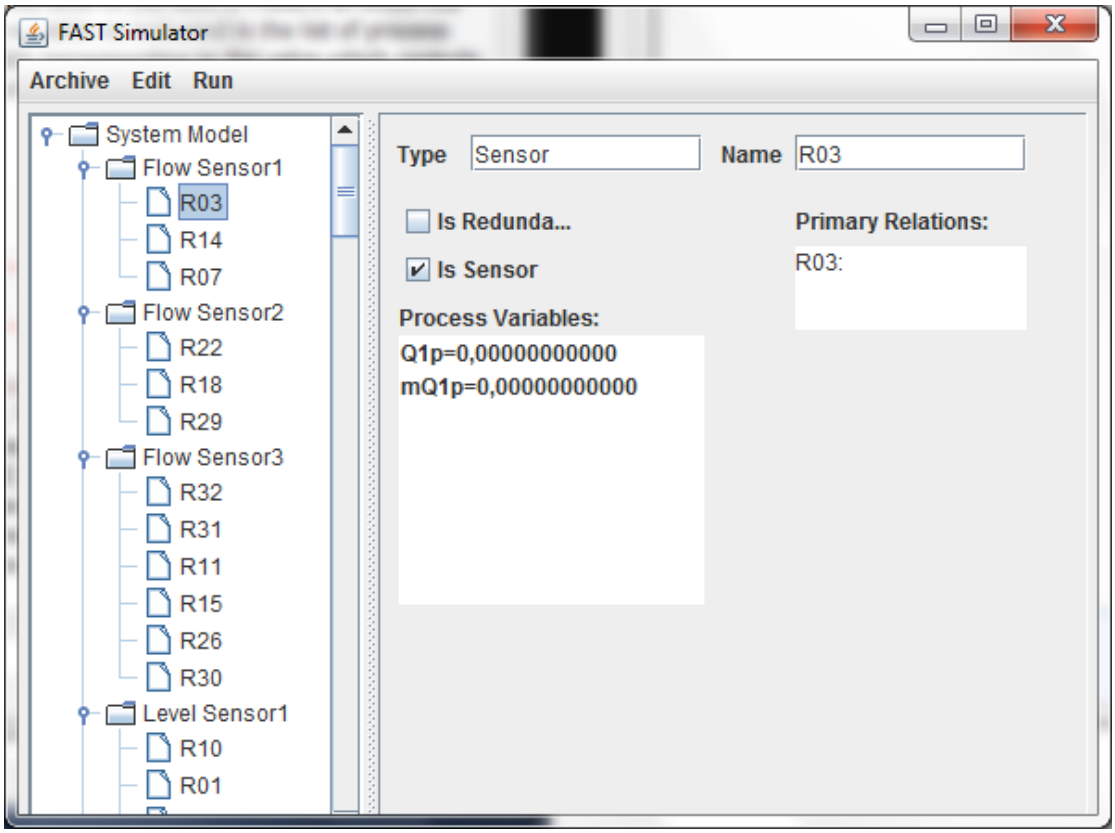


FIGURE A.2: FAST Simulator main window

After processing the PDL the application shows a tree view in the left panel with all components and their associated relations as childs (see Figure A.2). By selecting a relation it is possible to display the following relation attributes in the right panel:

- Type of the relation. Refer to Section A.2 for a list of possible relation types.
- Name: Name of the relation as indicated in the PDL.
- Redundant: If checked indicates that the relation is redundant.
- Primary Relations: If the relation is redundant, indicates the list of primary relations used to solve the not measured process variables in this relation.
- Process Variables: Displays the process variables involved in that relation and the current value updated during the simulation.

The 'comma-separated-values' (csv) files can be opened by several generic software applications, from MATLAB to the Microsoft Excel.

The figure A.3 show the structural matrix file 'sctruct_matrix.csv' opened and formatted with Microsoft Excel.

The figure A.4 show the perfect matching matrix file 'fault_signature_matrix.csv' opened and formatted with Microsoft Excel.

The figure A.5 show the fault signature matrix file 'perfect_matching.csv' opened and formatted with Microsoft Excel.

Proc Vars	mH1	mH2	mQp	mUb	mUo	mUp	H1	H2	Q12	Qo	Qp	Ub	Uo	Up	dH1	dH2
R01	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
R02	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
R03	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
R04	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
R05	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
R06	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
R07	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
R08	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
R09	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0
R10	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
R11	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0
R12	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0
R13	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1
R14	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
R15	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

FIGURE A.3: Structural Matrix file generated by the FAST Simulator

A.3.2 Simulating the Process

By selecting from the menu the option 'Run/Start' a dialog will be displayed allowing to simulate the process providing the simulation time and the simulation step (equivalent to the sample period). After confirming the simulation of the process will start. The simulation generates the following files with the results:

- Simulation Output file ('simoutput.csv'): a comma separated values file with the values of each of the relations at each simulation step. With this file it is easy to obtain a graph of the behaviour of the different relations.
- Residuals output file ('arroutput.csv'): a comma separated values file with the values of each of the residuals at each simulation step. With this file it is possible to generate a graph of each of the residuals.
- Error output file ('erroutput.csv'): a comma separated values file with the result of the residual evaluation.

As with the files generated during the structural analysis at loading the PDL, *FAST* generates these files in a format which can be opened from many generic software applications suchs as MATLAB or the Microsoft Excel.

The figure A.6 is an example of the simulation outputs when simulating the two-tanks example obtained from the file 'simoutput.csv' after some formatting in Excel, selecting all relation values and using the option Insert Graph. It can be seen how the tank levels identified by the relations R01 (Tank1 level) and R02 (Tank2 level) reach the set point and how the R04 corresponding to the valve communicating the two tanks changes the state to 0 when the set-point level of the Tank2 is reached.

The figures A.7, A.8, A.9, A.10, A.11, show how using the same Microsoft Excel funcionality it is easy to draw the residual graphs using the file 'arrouput.csv'. The file contains for each residual 4 columns:

- Residual Value: The value of the residual at each simulation step.
- Residual Average: The average of the value calculated from the window of N samples.

Proc Vars	mH1	mH2	mQp	mUb	mUo	mUp	H1	H2	Q12	Qo	Qp	Ub	Uo	Up	dH1	dH2
R01	1	0	0	0	0	0	1*	0	0	0	0	0	0	0	0	0
R02	0	1	0	0	0	0	0	1*	0	0	0	0	0	0	0	0
R03	0	0	1	0	0	0	0	0	0	0	1*	0	0	0	0	0
R04	0	0	0	0	0	1	0	0	0	0	0	0	0	1*	0	0
R05	0	0	0	1	0	0	0	0	0	0	0	1*	0	0	0	0
R06	0	0	0	0	1	0	0	0	0	0	0	0	1*	0	0	0
R07	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
R08	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
R09	0	0	0	0	0	0	1	1	1*	0	0	1	0	0	0	0
R10	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
R11	0	0	0	0	0	0	0	1	0	1*	0	0	1	0	0	0
R12	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1*	0
R13	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1*
R14	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
R15	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

FIGURE A.4: Perfect Matching matrix file generated by the FAST Simulator

Residual	Flow Sensor1	Level Sensor1	Level Sensor2	On/Off Controller	PI Controller	Pump	Tank1	Tank2	Valve Output	Valve Tanks
z_1	1	0	0	0	1	1	1	0	0	0
z_2	0	1	0	0	1	1	1	0	0	1
z_3	0	0	1	1	0	0	0	1	1	1
z_4	1	1	1	1	1	1	1	1	1	1
z_5	0	1	1	1	1	0	1	1	1	1

FIGURE A.5: Structural Matrix file generated by the FAST Simulator

- Residual Deviation: The deviation calculated taking into account the $5 \cdot \sigma$ rule.

The file 'errouput.csv' it is the file resulting of the evaluation of the fault vector at each simulation step. The details of the evaluation process of the fault vector and the evaluation of the residuals are described in Section 4.3.

A.3.3 Simulating a Fault

With *FAST* it is possible to simulate additive and multiplicative faults in any of the relations. To add a fault, in the PDL include the following XML attribute in a relation:

```
<Error Type="[Type of error]">
     <Value>[Value]/Value>
     <TimeStart>[Time to start in seconds]</TimeStart>
     <TimeEnd>[Time to end in seconds]</TimeEnd>
</Error>
```

For example, to indicate an additive error in a level sensor, we can define the relation as follows:

```
<Relation Name="R02" Type="Sensor">
     <Component>Level Sensor2</Component>
     <ProcessVar>H2</ProcessVar>
     <ProcessVarMeasured>mH2</ProcessVarMeasured>
```

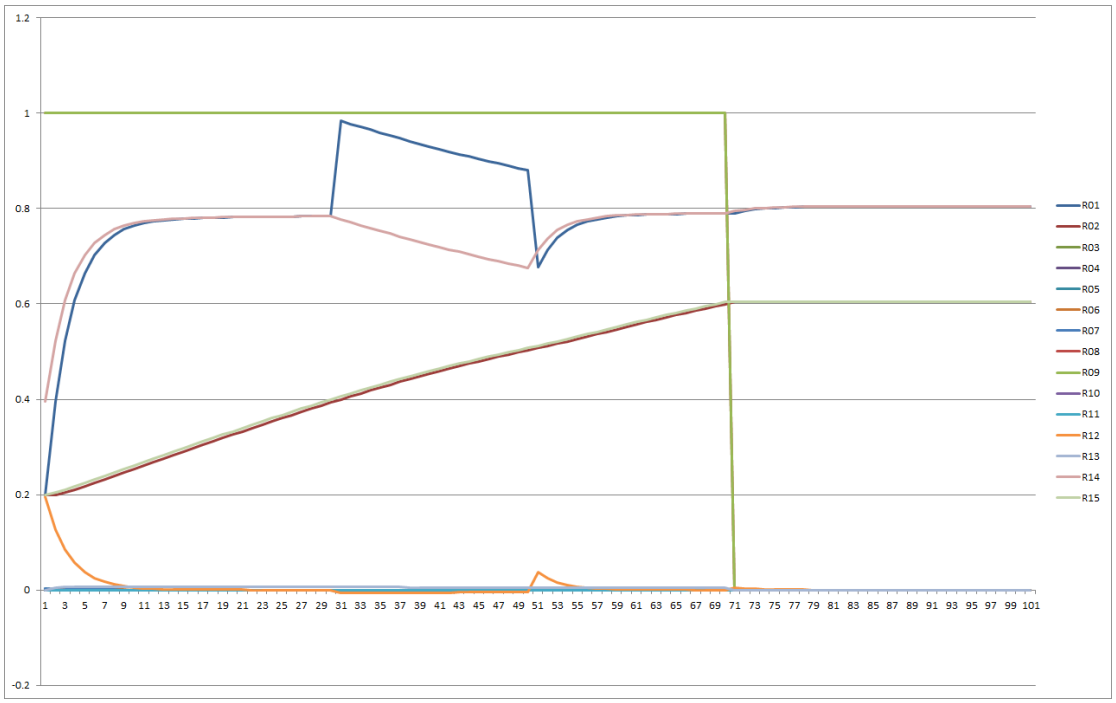


FIGURE A.6: Simulation of the two tanks process

```
      <Precision>0.001</Precision>
      <Error Type="Additive">
            <Value>1</Value>
            <TimeStart>30</TimeStart>
            <TimeEnd>40</TimeEnd>
      </Error>
</Relation>
```

In this case, the error will add 1 to the sensor output from the second 30 to the second 40 of the simulation.

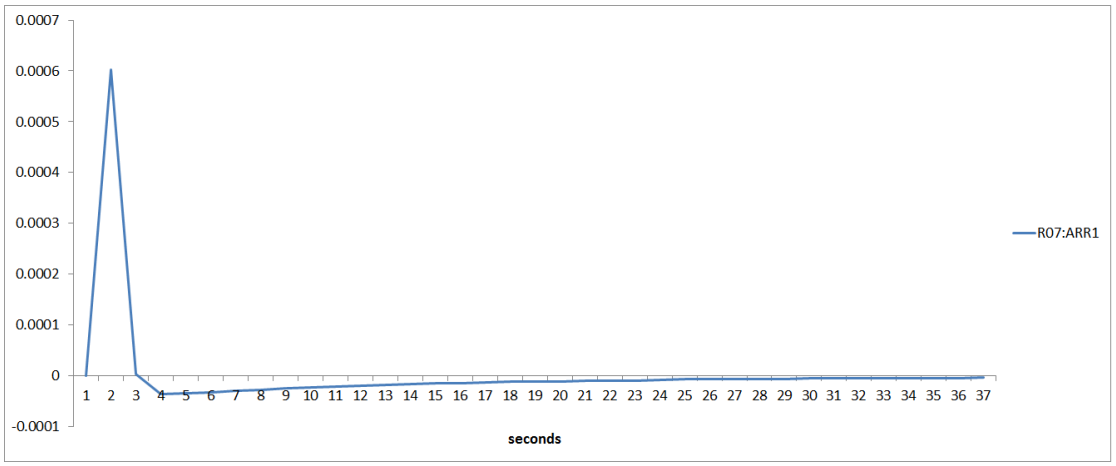


FIGURE A.7: Residual z_1 of the two tanks process

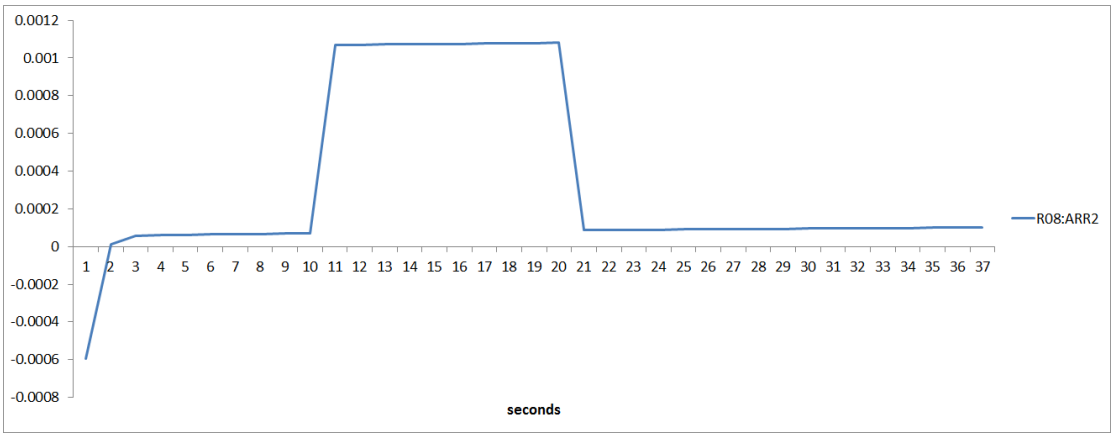


FIGURE A.8: Residual z_2 of the two tanks process

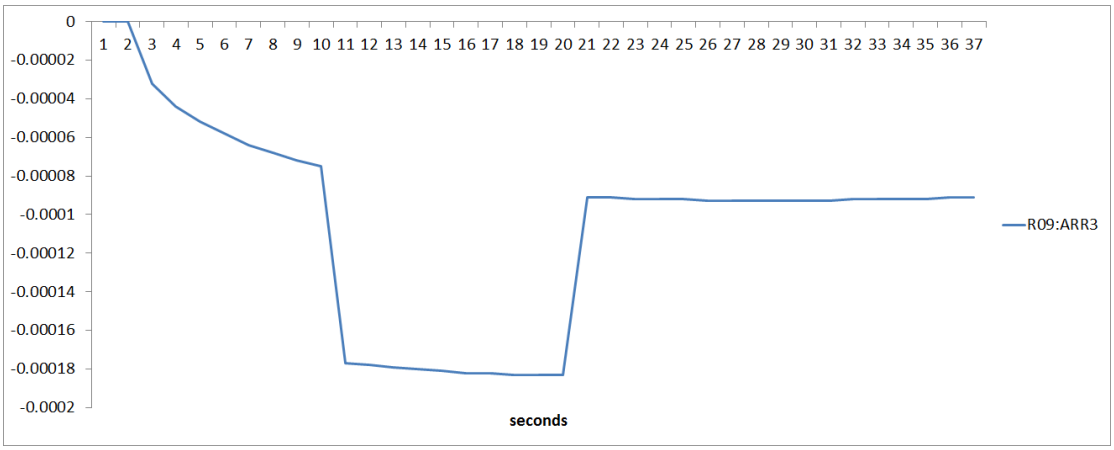


FIGURE A.9: Residual z_3 of the two tanks process

A.4 FAST Supervisor Agent

A.4.1 Installation Requirements

To set-up the on-line process supervision it is needed the following infraestructure:

- The JADEX Agent Framework
- An OPC Server
- A SCADA

The JADEX Agent Framework is a Java application, so the same constraints than for the *FAST* simulator apply. That is, wherever the Java Virtual Machine (JVM) can run, the JADEX Framework can be executed. In this case the computation requirements are more demanding, although not exhaustive tests have been performed, in a Windows 7 computer with 4 GB of RAM works without problems. The JADEX Agent Framework is the context in which the Supervision Agents are executed.

As indicated in Section 5.2, each agent can run in a different JVM, even in remote computers with Internet or TCP/IP connection. This functionality allows the distributed FDI implementation.

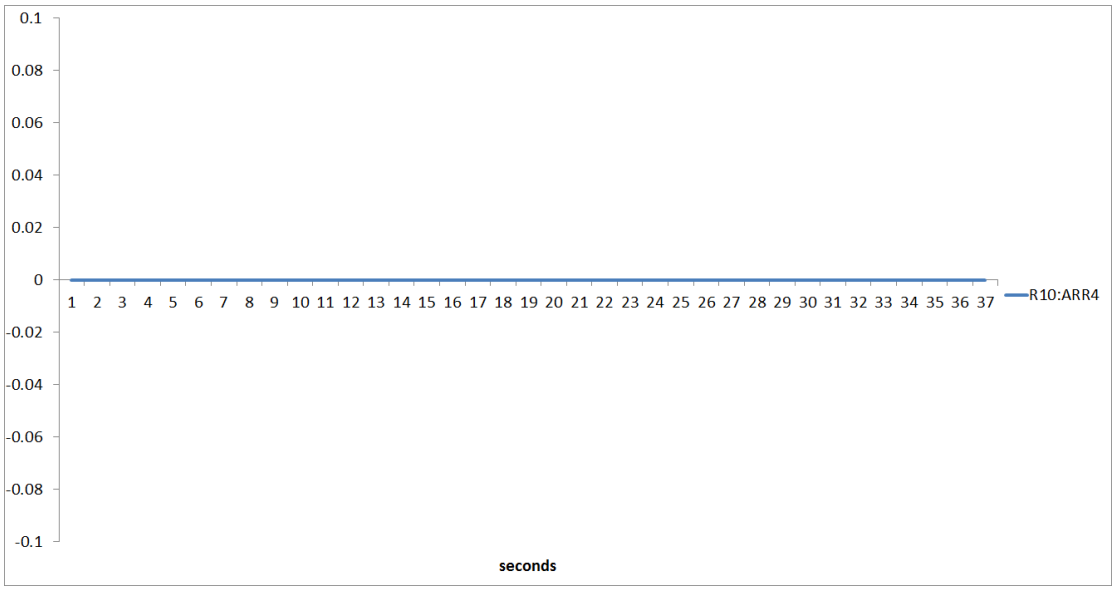


FIGURE A.10: Residual z_4 of the two tanks process

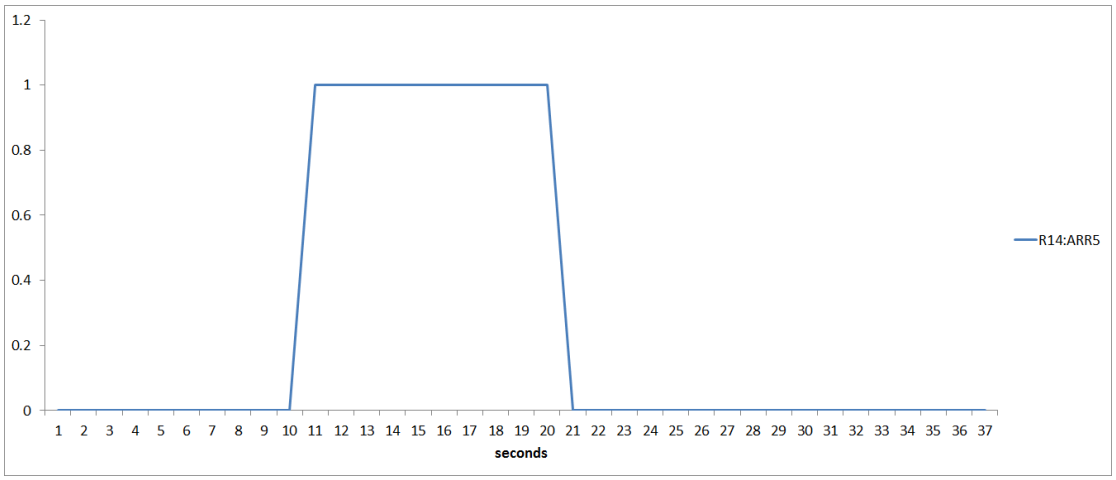


FIGURE A.11: Residual z_5 of the two tanks process

The OPC Server can be any product compliant with the OPC standard. In the experiments we have used the Matrikon OPC Server which is available for download from MatrikonOPC®. The OPC server requires to configure the tags used by the Supervision Agent. But this configuration is identical to the configuration of the tags for any SCADA system which accesses to the process via OPC.

The SCADA is the system where finally the results of the FDI process will be displayed. In the ideal case, and if a good identification is possible in the process, each component in the different SCADA displays will have an indicator if this component has a fault, being the indicator feed by the information provided by the Supervisor Agent. For the experiments in this work the General Electric iFix® SCADA software is used.

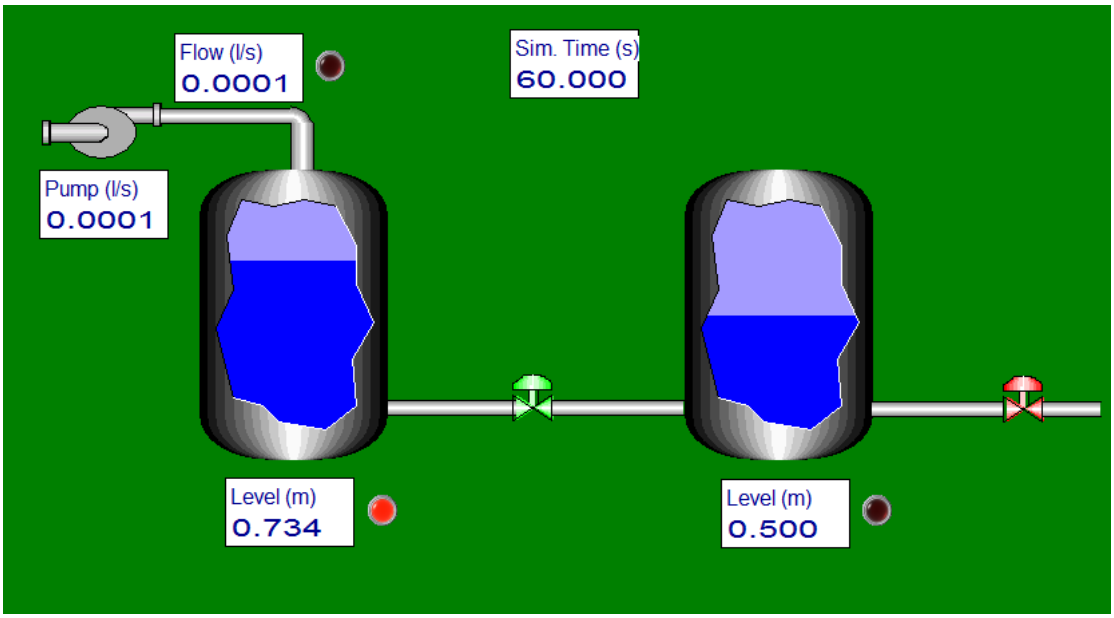


FIGURE A.12: The two tanks representation in a SCADA

A.4.2 Configuring the Supervision Agent

Each supervision agent has an Agent Definition File, which is also implemented in XML. In this agent definition file the most important parameter and in principal the only parameter which needs to be modified is the path to the Process Definition File to be used by this agent. This parameter is indicated as an agent 'believe' as can be seen below:

```
<belief name="PDL_file" class="String" exported="true">
     <fact>"D:\\iAgent\\Support files\\twotanks.xml"</fact>
</belief>
```

The Agent Definition File is loaded from the JADEX Agent Framework at run-time (see Listing **??**). The PDL contains the definition of the OPC Server to which the agent will be connected:

```
<Agents>
<Agent name="SA1" opc_server="Matrikon.OPC.Simulation.1"/>
<Agent name="SA2" opc_server="Matrikon.OPC.Simulation.1"/>
</Agents>
```

Note that in a distributed configuration the PDL indicates which components are supervised for each agent. This is also indicated in the PDL by setting the attribute 'agent' with the corresponding 'Agent name'.

LISTING A.2: JADEX Agent Definition File (ADF) for the Supervisor Agent

```
1 <agent xmlns="http://jadex.sourceforge.net/jadex"
2         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xsi:schemaLocation="http://jadex.sourceforge.net/jadex http://jadex.sourceforge.net/jadex-2.0.xsd"
4         name="ProcessSupervisor"
5         package="ProcessSupervisorAgent">
6         <imports>
7                 <import>java.util.logging.*</import>
8                 <import>java.util.*</import>
```

```
9               <import>jadex.adapter.base.fipa.*</import>
10          </imports>
11          <beliefs>
12              <belief name="environment" class="IEnvironment">
13                  <fact>Environment.getInstance()</fact>
14              </belief>
15              <belief name="PDL_file" class="String" exported="true">
16                  <fact>"D:\\Project_Files\\ampanet\\trunk\\iAgent\\Support_files\\twotanks_tankscale.xml"</
                        fact>
17              </belief>
18              <belief name="process" class="Supervisor.Process">
19                  <fact>$beliefbase.environment.loadProcessDefinitionFile($beliefbase.PDL_file)</fact>
20              </belief>
21          </beliefs>
22          <goals>
23              <!-- The goal to detect a fault. -->
24              <performgoal name="detect_faults" exclude="never" retrydelay="2000">
25              </performgoal>
26              <achievegoal name="notify_faults">
27              </achievegoal>
28          </goals>
29          <events>
30              <messageevent name="request_var" direction="receive" type="fipa">
31                  <parameter name="performative" class="String" direction="fixed">
32                      <value>jadex.bridge.fipa.SFipa.REQUEST</value>
33                  </parameter>
34              </messageevent>
35              <messageevent name="inform" direction="send" type="fipa">
36                  <parameter name="performative" class="String" direction="fixed">
37                      <value>jadex.bridge.fipa.SFipa.INFORM</value>
38                  </parameter>
39              </messageevent>
40            <internalevent name="gui_update">
41              <parameter name="content" class="String"/>
42            </internalevent>
43          </events>
44          <plans>
45              <!-- Reactive plan to calculate residuals -->
46              <plan name="calculate">
47                  <body class="CalculateResidualsPlan"/>
48                  <trigger>
49                      <goal ref="detect_faults"/>
50                  </trigger>
51              </plan>
52              <!-- Reactive plan to notify a fault -->
53              <plan name="notify">
54                  <body class="SendFaultPlan"/>
55                  <trigger>
56                      <goal ref="notify_faults"/>
57                  </trigger>
58              </plan>
59              <plan name="provide_var">
60                  <body class="ProvideVarPlan"/>
61                  <trigger>
62                      <messageevent ref="request_var"/>
63                  </trigger>
64              </plan>
65              <!-- Initial plan for creating and updating the gui. -->
66              <plan name="gui">
67                  <body class="GuiPlan"/>
68              </plan>
69          </plans>
70          <events>
71              <!-- Specifies an internal event for updating the gui.-->
72              <internalevent name="gui_update">
73                  <parameter name="content" class="String[]"/>
74              </internalevent>
75          </events>
76          <expressions>
77          </expressions>
78          <configurations>
79              <configuration name="default">
80                  <goals>
81                      <!-- Initial goal for searching for garbage. -->
82                      <initialgoal ref="detect_faults"/>
83                  </goals>
84                  <plans>
85                      <initialplan ref="gui"/>
86                  </plans>
87              </configuration>
88          </configurations>
89  </agent>
```

A.4.3 The JADEX Framework

The JADEX Agent Framework can be started from the command line. However, the path to the different libraries used by the Supervision Agent needs to be defined. A tutorial of this framework is available in Braubach and Pokahr, 2007. The paths need

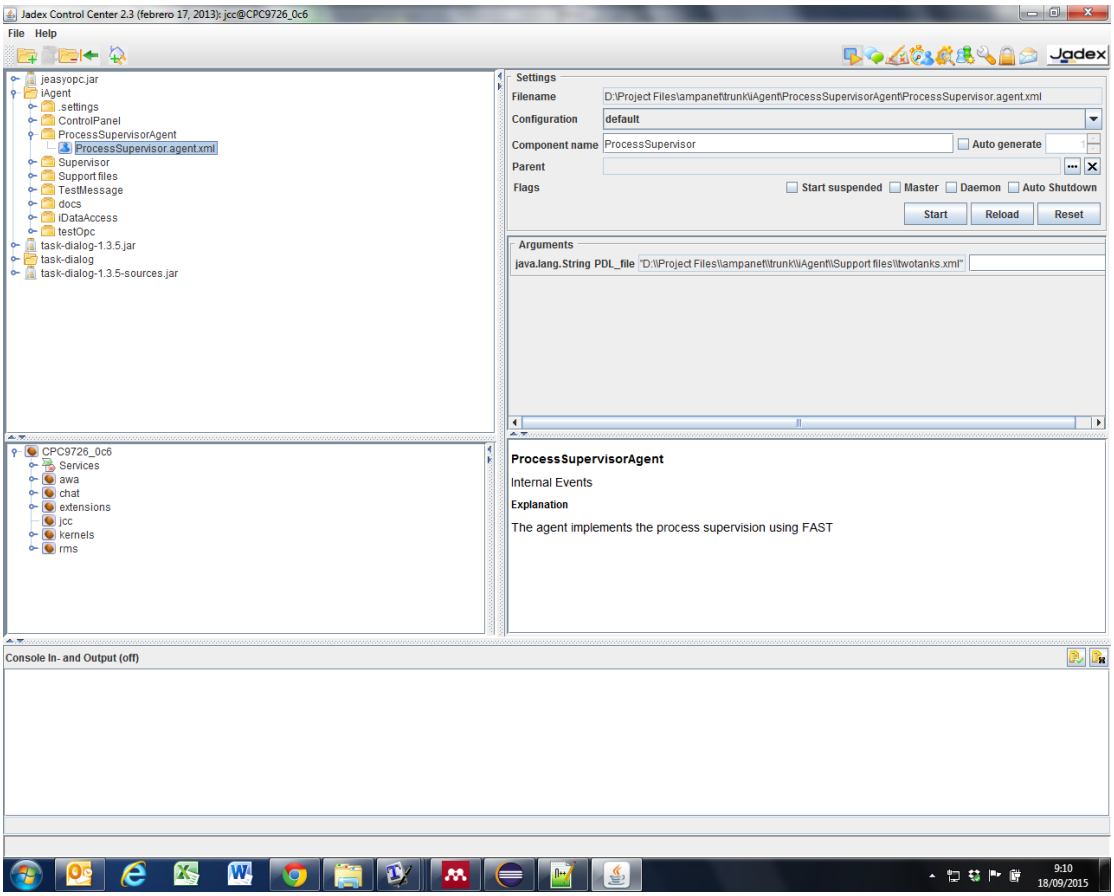


FIGURE A.13: JADEX Agent Framework

to be added using the JADEX framework main window as can be seen in the figure A.13.

The agent is loaded in the framework from the Start window (main window in the JADEX Framework) by selecting the Agent Definition File in the tree on the left panel and pressing the Start button on the right panel. If the agent is loaded successfully, it will start communicating with the OPC Server acquiring process data, calculating the fault vector and evaluating the fault vector with respect to the process components indicated in the PDL. In case a fault is detected it will provide as well the information to the OPC Server tags defined in the PDL for this purpose. The SCADA, which needs to be configured to include in the synoptic displays these tags, it will show the fault information when generated.

It is possible to communicate with any Supervisor Agent from the JADEX Agent Framework from the Conversation Center. In this window it is possible to send messages to the agent to acquire the value of a process variable by sending the message:

```
'request([name of the process variable])'
```

The agent can be in the local computer or in a remote computer if accessible through Internet or TCP/IP. The server/computer name is part of the agent name as can be seen in the right panel window of the Conversation Center in the figure A.14. If the agent is in a remote computer the agent name needs to contain the IP Address or the URL (Unified Resource Location) computer name. In fact the Conversation Center is acting as any other Agent in a Multi-agent system.

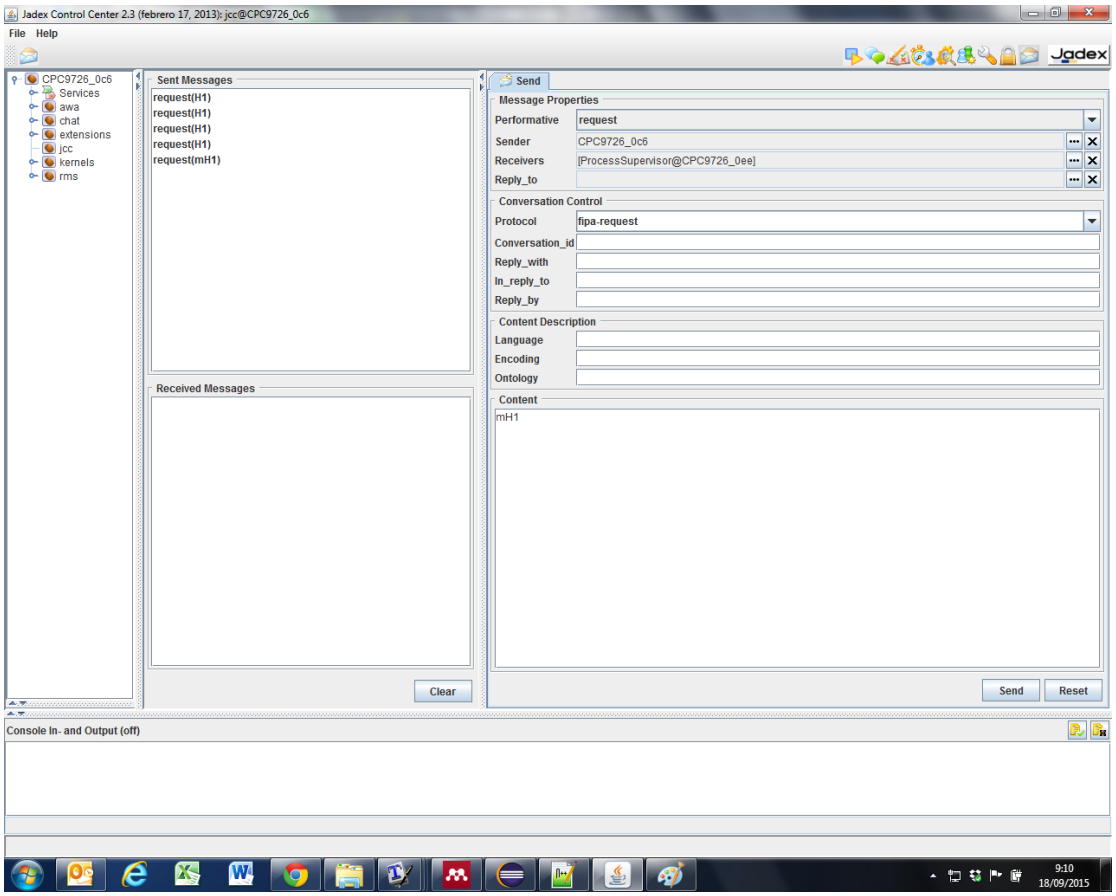


FIGURE A.14: JADEX Conversation Center

A.4.4 Working with the *FAST* Simulator

The architecture of *FAST* allows the testing of the Supervisor Agent using simulated data from the *FAST* simulator. The Simulator is able to feed the OPC server with simulated process data and the Process Supervisor Agent is able to read this data and process it. In this way it is possible to close the loop, that is, simulate a fault with the simulator and see the result of this fault in a SCADA generated by the FDI processing implemented in the Supervisor Agent, see Figure A.15.

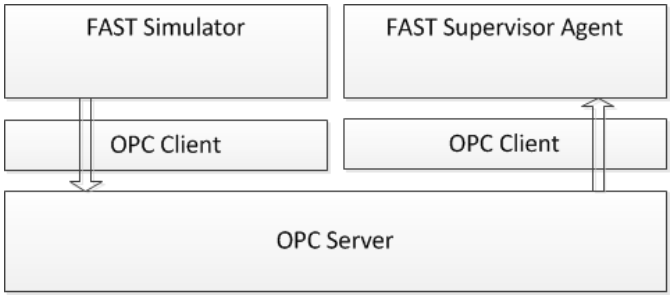


FIGURE A.15: *FAST* Simulator and Supervisor Agent connected

Appendix B

FAST Model Extension

In this Section, it will be presented the way of generating the model of a new component in *FAST*. The models in *FAST* are implemented as Java classes. The classes need to have a predefined structure and some mandatory methods. In *FAST* to add a new component apart of generating the Java classes for the corresponding relations, the parsing module needs to be modified. An opportunity of improvement is to generate component definition files which could be parsed by the tool which defines the structure of the component and its relations and then avoid the necessity of modifying the parser.

B.1 Creating a New FAST Component Model

In this Section, the process of creating a new component model will be described. In order to perform properly this process, knowledge of the Java programming language is required. All component models in *FAST* are programmed as Java classes.

B.1.1 The TankScale Model

To illustrate the way of creating a new component, the component 'TankScale' and its corresponding relations will be generated. A TankScale is a tank were the level is obtained indirectly with a scale by measuring the mass in kilograms inside the reactor instead the level. The tank has an input flow and an output flow. The density of the liquid is known and therefore it is possible by using the flow input/output measurements to calculate the mass-in and mass-out from the tank. Therefore, by using the mass conservation principle it is possible to define the following relation for this component:

$$\frac{dM}{dt} = \rho(Q_i(t) - Q_o(t)) \tag{B.1}$$

being dM/dt the variation of the mass, ρ the liquid density, Q_i the flow input and Q_o the flow output.

The component will require the initial value of the mass of the tank to be provided in the process variable M corresponding to the tank mass.

The component will have as well associated a mass sensor, that is, the scale, but this relation can be reused, since it is a Sensor relation.

B.1.2 Creating the Component Java Class

The example will be described using Eclipse as the development environment, although the code can be programmed using any development environment or Java code

editor. The *FAST* models are located in the Supervisor package. In Eclipse select the Supervisor package and select Create a new class (see Figure B.1).

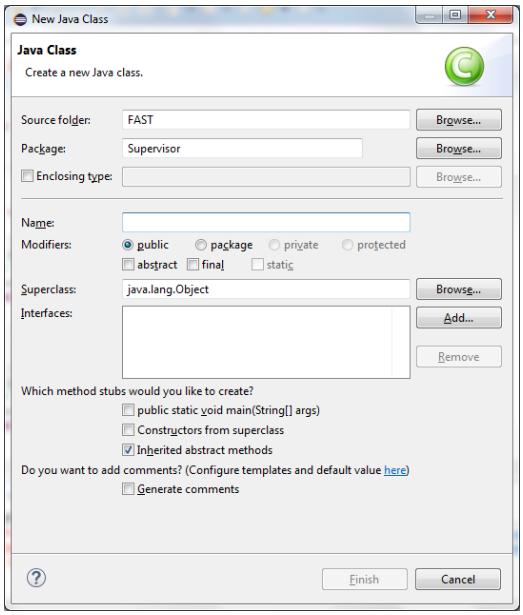


FIGURE B.1: Creating the class for a new model

- Input in the class Name: TankScale
- Input in the Superclass: Relation
- and press Finish.

The initially generated code for the TankScale component class is indicated in the Listing B.1.2:

```
1  /**
2   *
3   */
4  package Supervisor;
5
6  /**
7   * @author J.Duatis
8   *
9   */
10 public class TankScale extends Relation {
11
12         /* (non-Javadoc)
13          * @see Supervisor.Relation#output(double)
14          */
15         @Override
16         public double output(double timeStep) {
17                 // TODO Auto-generated method stub
18                 return 0;
19         }
20
21         /* (non-Javadoc)
22          * @see Supervisor.Relation#calcResidual()
23          */
24         @Override
25         public double calcResidual() throws RelationException {
26                 // TODO Auto-generated method stub
27                 return 0;
28         }
29
30         /* (non-Javadoc)
31          * @see Supervisor.Relation#calcProcessVar(Supervisor.Relation, Supervisor.ProcessVar)
32          */
33         @Override
34         public double calcProcessVar(Relation parentRel, ProcessVar var)
35                         throws RelationException {
```

```
36                  // TODO Auto-generated method stub
37                  return 0;
38          }
39
40  }
```

As the base class has no default constructor, it is mandatory to define a specific constructor. The constructor will be used to instantiate the class from the parser module and it shall have two parameters:

- String name: parameter receiving the name as indicated in the PDL.
- Component c: reference to the component to which this relation will belong.

The code of the constructor is indicated in the Listing B.1.2.

```
1  TankScale(String name, Component c)
2  {
3          super(name, c);
4  }
```

The following step will be to define the content of the method `output`. This method shall contain the calculation of the relation. It will be used during the simulation to provide the value of the relation obtaining the value of the process variables from the measured values. This relation will manage three process variables:

- Q_i as the flow input.
- Q_o as the flow output.
- dM as the derivative of the tank mass.

Therefore, it is practical to define an attribute for each of these process variables and the corresponding methods to assign the process variable references and another attribute for the density parameter ρ:

```
1  ProcessVar Qi;
2  ProcessVar Qo;
3  ProcessVar dM;
4  double param_rho;
5  ...
6
7  public void setProcessVarFlowIn(ProcessVar pv)
8  {
9          Qi = pv;
10         addProcessVar(pv);
11         setCausal(pv, true);
12 }
13
14 public void setProcessVarFlowOut(ProcessVar pv)
15 {
16         Qo = pv;
17         addProcessVar(pv);
18         setCausal(pv, true);
19 }
20
21 public void setProcessVarDeltaMass(ProcessVar pv)
22 {
23         dM = pv;
24         addProcessVar(pv);
25         setCausal(pv, true);
26 }
27
28 public void setDensity(double rho)
29 {
30         param_rho = rho;
31 }
```

The method `addProcessVar` adds the process variable to the list of process variables managed in this relation and the method `setCausal` indicates that the process variable is causal for this relation, which means that it is possible to obtain the value of this process variable knowing the value of the rest. Once this intermediate steps have been implemented it is easy to define the `output` method as:

```
public double output(double timeStep) {
        double result = (Qi.getValue() -
                Qo.getValue()) * param_rho;
        result = getErrorValue(result);
        dM.setValue(result);
        return result;
}
```

The method `getErrorValue(result)` is used to be able to introduce process errors. In the PDL it is possible to indicate an additive or multiplicative error during a period of time, and taking the result as input it will apply to the value the corresponding error if indicated. The `ProcessVar` class has the method `setValue` to assign the current value.

The method `CalcResidual` shall be implemented to cover the case that this relation is selected as a redundant relation and a residual is derived. In this case, all process variable values need to be obtained from other relations and all terms of the relation passed to one hand of the equal, forcing the relation to evaluate 0, that is, converting the relation to a residual which if there is no fault it will provide as a result a value near to 0.

The method which provides the value of a process variable from other relations is `calcFromRelation`, and as parameters, the `this` and the process variable shall be indicated. The function is implemented below:

```
public double calcResidual() throws RelationException {
        double residual = 0;
        double vdM = calcFromRelation(this, dM);
        double vQi = calcFromRelation(this, Qi);
        double vQo = calcFromRelation(this, Qo);

        residual = vdM / param_rho - vQi + vQo;

        return residual;
}
```

Finally it is possible that the relation, as a result of the Structural Analysis is selected as an elementary relation. In this case, some process variable will be required to be calculated but obtaining the value of the other process variables from other relations, as indicated in the Structural Matrix. To implement this functionality the class shall implement the method `calcProcessVar`. This method will receive the parent relation requiring the value of the process variable and the reference to the process variable itself. The method is illustrated in the listing below:

```
public double calcProcessVar(Relation parentRel, ProcessVar var) throws RelationException
{
        double result = 0;
        if (var == Qo)
                result = calcFlowOutput(parentRel);
        else if (var == Qi)
                result = calcFlowInput(parentRel);
        else if (var == dM)
                result = calcDerivative(parentRel);
        else
        {
                String s = String.format("Variable %s cannot be calculated from relation %s",
                        var.getSymbol(), this.getName());
                throw new RelationException(s);
        }
        return result;
}
```

The methods `calcFlowOutput, calcFlowInput, calcDerivative` are private methods which shall be implemented to calculate the value of the indicated process variables using the values of the other variables obtained from other relations:

```
private double calcFlowOutput(Relation parentRel) throws RelationException
{
        double vdM = parentRel.calcFromRelation(this, dM);
        double vQi = parentRel.calcFromRelation(this, Qi);
        double result = 0;
        result = vdM / param_rho - vQi;
        return result;
}
```

```
9
10 private double calcFlowInput(Relation parentRel) throws RelationException
11 {
12         double vdM = parentRel.calcFromRelation(this, dM);
13         double vQo = parentRel.calcFromRelation(this, Qo);
14         double result = 0;
15         result = vdM / param_rho + vQo;
16         return result;
17 }
18
19 private double calcDerivative(Relation parentRel) throws RelationException
20 {
21         double vQi = parentRel.calcFromRelation(this, Qi);
22         double vQo = parentRel.calcFromRelation(this, Qo);
23         double result = 0;
24         result = (vQi-vQo)*param_rho;
25         return result;
26 }
```

Two important things to note from these methods: The first is that the method `calcFromRelation` is called from the `parentRel` object which is the parent relation. This is because the process variables need to be calculated from the set of relations belonging to the parent relation. This set is determined when performing the Structural Analysis. The second important thing is that the method can return the `RelationException` indicating that there is some error in the process because the process variable required cannot be calculated from this relation. This is a bad thing, because if this method is requested indicates that the Structural Analysis has determined that the process variable can be calculated from this relation, therefore some coding problem should be present somewhere.

And with this steps we have finalized the class representing the model of a TankScale. When we introduce this model in the process it will probably generate some analytical redundancy since in case there are flow sensors and the scale sensor providing the mass, the relation can be used to calculate any of the process variables, since all will be measured.

B.1.3 Implementing the Parsing of the PDL

The following step will be to implement the parsing part in the parser module to read the parameters from the PDL.

In the Supervisor package edit the class DomParser. This class is the parser module and contains the parsing of all the components, process variables and relations represented in the PDL. To add a new relation type, the following methods shall be modified:

- getRelation: method which instantiates the relation type by calling the method to parse the specific relation parameters.
- newTankScaleRelation: new method to be created to parse the specific parameters of this relation.

The `getRelation` method needs to be modified by adding a new line to instantiated the new relation. See the code listing below:

```
1 ...
2 if (type.equalsIgnoreCase("TankScale")) rel = newTankScale(name, relEl);
3 ...
```

Following the method `newTankScale` needs to be implemented.

```
1  private Relation newTankScale(String name, Element relEl) throws ProcessParserException
2  {
3          Component c = getComponent(getTextValue(relEl,"Component"));
4          TankScaleRelation tr = new TankScaleRelation(name, c);
5          tr.setProcessVarDeltaMass(getProcessVar(getTextValue(relEl, "ProcessVarDeltaMass")));
6          tr.setProcessVarFlowIn(getProcessVar(getTextValue(relEl,"ProcessVarFlowIn")));
7          tr.setProcessVarFlowOut(getProcessVar(getTextValue(relEl,"ProcessVarFlowOut")));
8          tr.setDensity(getDoubleValue(relEl,"Density"));
9          return tr;
10 }
```

An example of the corresponding PDL representation in XML is illustrated in the following listing:

```
<Relation Name="R15" Type="TankScale">
        <Component>Tank3</Component>
        <ProcessVarDeltaMass>dM3</ProcessVarDeltaMass>
        <ProcessVarFlowIn>Q13</ProcessVarFlowIn>
        <ProcessVarFlowOut>Qo</ProcessVarFlowOut>
        <Density>998.2071</Density>
</Relation>
```

B.2 Simulating the New Component Model

In this Section, the newly generated component will be integrated in a process model and tested as part of the FDI analysis of that process. In order to ease the integration of the new model into the already presented two-tanks system another relation is needed which translates the measure in mass into a level. This new relation has been modeled as the *LevelScaleSensor*. The relation requires the density and the tank area to implement the translation. The relation will then calculate the process variable H_2 and the measured variable H_{m2} used by other relations.The PDL representation of this relation is:

```
<Relation Name="R15" Type="ScaleLevelSensor">
        <Component>Level/Scale Sensor</Component>
        <ProcessVarScaleMeasured>mM2</ProcessVarScaleMeasured>
        <ProcessVarLevel>H2</ProcessVarLevel>
        <ProcessVarLevelMeasured>mH2</ProcessVarLevelMeasured>
        <Density>998.2071</Density>
        <TankArea>0.0154</TankArea>
</Relation>
```

In the PDL there will be then 16 relations, one more than for the PDL indicated in the Section 4.4 as the new relation *ScaleLevelSensor* has been added. The relations are indicated in table B.1.

After loading the PDL into the *FAST* simulator the Structural Matrix is generated with the indicated relations belonging to the perfect matching. The Structural Matrix is indicated in table B.2.

TABLE B.1: Scale Tank example component relations

Relation	Type	Component	Variables	Description
$\langle r_1 \rangle$	Sensor	Level Sensor 1	H_{m1}, H_1	Level sensor tank 1
$\langle r_2 \rangle$	Sensor	Mass/Level Sensor	M_m, M	Mass sensor of tank 2
$\langle r_3 \rangle$	ScaleLevelSensor	Mass/Level Sensor	M_m, H_2, H_{m2}	Level of tank 2 obtained from the scale measurement
$\langle r_4 \rangle$	Sensor	Flow Sensor	Q_{mp}, Q_p	Flow sensor at pump output
$\langle r_5 \rangle$	Sensor	Pump	U_{mp}, U_p	Pump control signal
$\langle r_6 \rangle$	Sensor	Valve 1	U_{mb}, U_b	Two tanks linking valve control signal
$\langle r_7 \rangle$	Sensor	Valve 2	U_{mo}, U_o	Tank 2 output valve control signal
$\langle r_8 \rangle$	Pump	Pump	Q_p, U_p	Pump action
$\langle r_9 \rangle$	PI Controller	PI Controller	H_{m1}, U_p	PI controller
$\langle r_{10} \rangle$	Valve2Levels	Valve Tanks	Q_{12}, H_1, H_2, U_b	Calculated flow output from the valve linking the two tanks
$\langle r_{11} \rangle$	On-Off Controller	On-Off Controller	H_{m2}, U_b	Controller opening/closing the linking the two tanks
$\langle r_{12} \rangle$	Valve Level	Valve 2	Q_o, H_2, U_o	Calculated flow output from the tank 2 output valve
$\langle r_{13} \rangle$	Tank	Tank 1	$Q_p, Q_{12}, H_{m1}, \dot{H}_1$	Relation between flow input/output and tank 1 level
$\langle r_{14} \rangle$	Tank	Tank 2	$Q_o, Q_{12}, M, \dot{M}$	Relation between flow input/output and tank 2 mass
$\langle r_{15} \rangle$	Derivative	Tank 1	$H_1, \dot{H}_1$	Derivative level relation in tank 1
$\langle r_{16} \rangle$	Derivative	Tank 2	$M, \dot{M}$	Derivative mass relation in tank 2

TABLE B.2: Structural Matrix indicating the Perfect Matching with '*'

Relations	H_{m1}	H_{m2}	M_m	Q_{mp}	U_{mb}	U_{mo}	U_{mp}	H_1	H_2	M	Q_{12}	Q_o	Q_p	U_b	U_o	U_p	$\dot{H}_1$	$\dot{M}$
$\langle r_1 \rangle$	1	0	0	0	0	0	0	1*	0	0	0	0	0	0	0	0	0	0
$\langle r_2 \rangle$	0	0	1	0	0	0	0	0	0	1*	0	0	0	0	0	0	0	0
$\langle r_3 \rangle$	0	1	1	0	0	0	0	0	1*	0	0	0	0	0	0	0	0	0
$\langle r_4 \rangle$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1*	0	0	0
$\langle r_5 \rangle$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
$\langle r_6 \rangle$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1*	0	0
$\langle r_7 \rangle$	0	0	0	1	0	0	0	0	0	0	0	0	1*	0	0	0	0	0
$\langle r_8 \rangle$	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
$\langle r_9 \rangle$	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
$\langle r_{10} \rangle$	0	0	0	0	1	0	0	0	0	0	0	0	0	1*	0	0	0	0
$\langle r_{11} \rangle$	0	0	0	0	0	0	1	1	1	0	1*	0	0	1	0	0	0	0
$\langle r_{12} \rangle$	0	0	0	0	0	0	0	0	1	0	0	1*	0	0	1	0	0	0
$\langle r_{13} \rangle$	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
$\langle r_{14} \rangle$	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
$\langle r_{15} \rangle$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1*	0
$\langle r_{16} \rangle$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1*

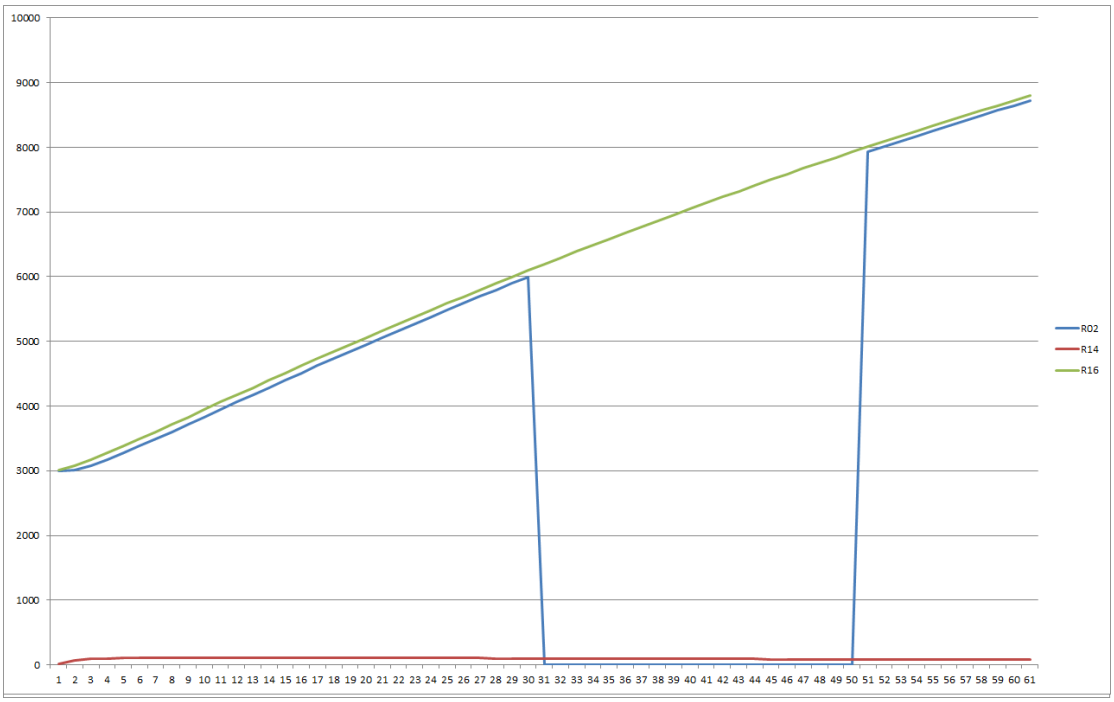


FIGURE B.2: Scale sensor error simulation.

In the same way, the fault signature matrix is obtained and indicated in table B.3.

TABLE B.3: Scale tank example fault signature matrix

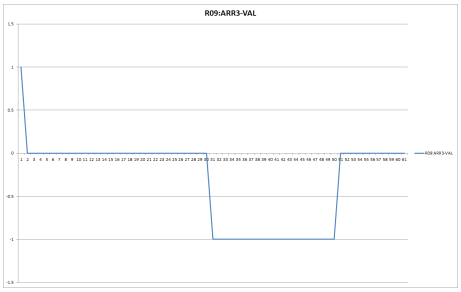
FSM	Q_{mp}	H_{m1}	M_{m2}	U_{mb}	U_{mp}	Pump	Tank 1	Tank 2	Valve Out	Valve Tanks
$z_1\langle r_7\rangle$	0	1	0	0	1	0	1	0	0	0
$z_2\langle r_8\rangle$	1	0	0	0	1	1	0	0	0	0
$z_3\langle r_9\rangle$	0	0	1	1	0	0	0	0	0	0
$z_4\langle r_{12}\rangle$	1	1	1	1	0	0	1	0	0	1
$z_5\langle r_{13}\rangle$	0	1	1	1	0	0	1	1	1	1

The simulation of the process, with a fault in the scale sensor which sets the value 0 in the interval from 30 to 50 seconds, can be seen in the figure B.2. In the figure only the measured scale sensor M_m relation $< r_2 >$ and the model variable M obtained from relations $< r_{14} >$ and $< r_{16} >$ is displayed, as the rest of relations have the same values as in the example indicated in Section 4.4.

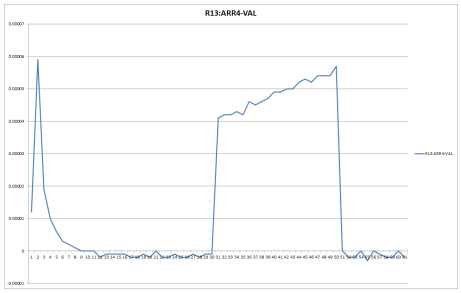
The residuals as generated during the simulation are presented in figure B.3.

B.3 Using a New FAST Component Model On-line

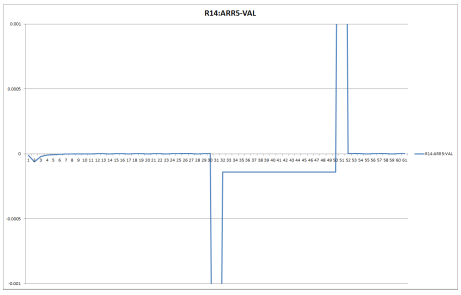
To test the component model Tank Scale on-line, it is possible to use the *FAST Simulator* connected to the OPC Server. The simulator is used to generate the process data which is sent to the OPC Server and the Supervisor Agent works as connected to a real process. The simulator is configured to update the process data in the OPC Server every second and the Supervisor Agent is configured to acquire and compute the residuals every 2 seconds. In the PDL the error simulation is set to start in the second 40 until the second 60.



(A) Residual z_3



(B) Residual z_4



(C) Residual z_5

FIGURE B.3: $LevelSensor_1$ fault residuals

The *FAST Simultor* is started and the JADEX environment is started as well. The Supervisor Agent starts acquiring the process data through the embedded OPC client and calculating the residuals. After some seconds after the second 40 from the simulation it can be seen in the Supervisor Agent monitor window how the tags LS_1002_01_ERR and CT_1002_01_ERR change to Value=true indicating an error in the Scale/Level sensor 2 or the Valve on/off controller (see Figure B.4). In this case, as can be seen in the Fault Signature Matrix (see Table B.3) the two components are sensitive to the same three residuals. The Supervisor Agent updates this Tags which can be displayed in a SCADA to indicate a possible error in any of these components.

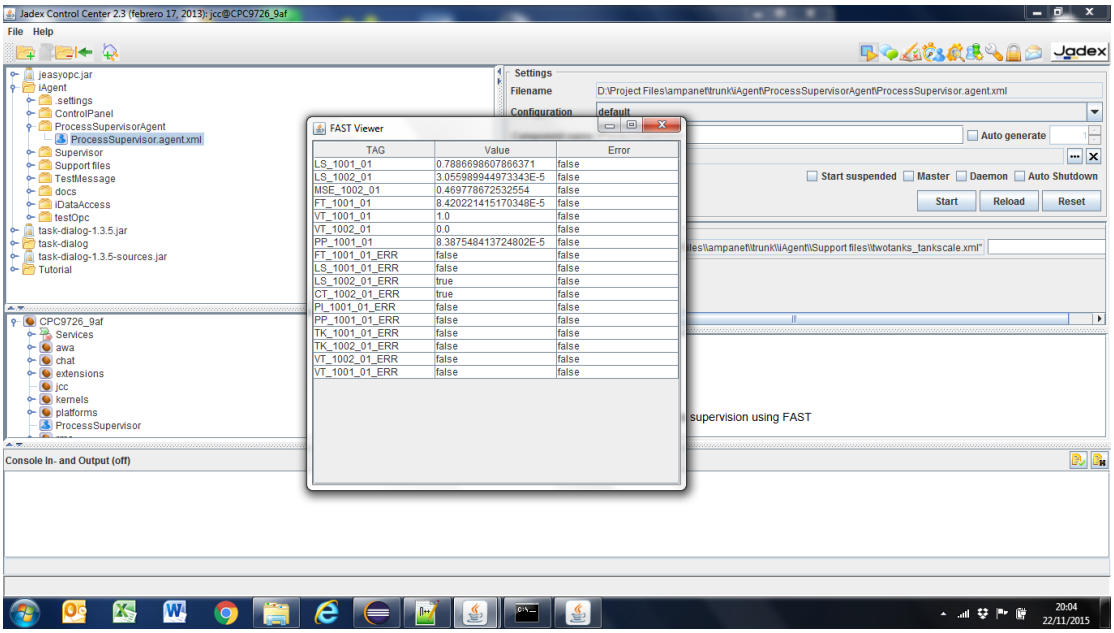


FIGURE B.4: Scale sensor error detected by the Supervisor Agent.

Bibliography

AOS. *JACK Documentation*. URL: http://www.agent-software.com.au/products/jack/documentation_and_instructi/jack_documentation.html (visited on 10/11/2015).

Belkacem, Anne-lise Gehin and Ould Bouamama (2015). "Design of distributed Fault Detection and Isolation systems". In: *9th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes.* Paris, France.

Binot, R a, C Tamponnet, and Christophe Lasseur (1994). "Biological life support for manned missions by ESA." In: *Advances in space research : the official journal of the Committee on Space Research (COSPAR)* 14.11, pp. 71–74. ISSN: 02731177. DOI: 10.1016/0273-1177(94)90281-X.

Blanke, Mogens and Torsten Lorentzen (2006). "SATOOL - A Software Tool for Structural Analysis of Complex Automation Systems". In: *6th IFAC Symposium on Fault Dection, Supervision and Safety of Technical Processes* 431, pp. 673–678.

Blanke, Mogens et al. (2006). *Diagnosis and Fault-Tolerant Control With contributions by Jochen Schröder*. 2nd Editio. Springer-Verlag Berlin Heidelberg, Chapter 5. ISBN: 978-354-035-6523.

Blesa, Joaquim et al. (2014). "FDI and FTC of wind turbines using the interval observer approach and virtual actuators/sensors". In: *Control Engineering Practice* 24, pp. 138–155. ISSN: 09670661. DOI: 10.1016/j.conengprac.2013.11.018. URL: http://linkinghub.elsevier.com/retrieve/pii/S096706611300230X.

Bouamama, Ould et al. (2005). "Model builder using functional and bond graph tools for FDI design". In: *Control Engineering Practice* 13.7, pp. 875–891. ISSN: 09670661. DOI: 10.1016/j.conengprac.2004.10.002.

Bratman, Michael E. (1999). *Intention, Plans, and Practical Reason*. Vol. 100. 2. University of Chicago, p. 277. ISBN: 9781575861920.

Braubach, L, W Lamersdorf, and Alexander Pokahr (2003). "Jadex : Implementing a BDI-Infrastructure for JADE Agents". In: 3.September, pp. 76–85.

Braubach, Lars and Alexander Pokahr (2007). "Jadex Tutorial". In: June.

Caire, Giovanni (2007). "Jade Programmer ' S Guide". In: C, pp. 1–53.

Cassar, J. P., Marcel Staroswiecki, and P. Declerck (1994). "Structural Decomposition of Large Scale Systems for the Design of Failure Detection and Isolation Procedures". In: *Systems Science* 20.1, pp. 31–42.

Chen, Jie (1995). "Model-Based Fault Diagnosis of Dynamic Systems". PhD Thesis. University of York.

Chen, Jie and Hongyue Zhang (1991). "Robust detection of faulty actuators via unknown input observers". In: *International Journal of Systems Science* 22.10, pp. 1829–1839. ISSN: 0020-7721. DOI: 10.1080/00207729108910753. URL: http://www.tandfonline.com/doi/abs/10.1080/00207729108910753.

Chow, E. and A. Willsky (1984). "Analytical redundancy and the design of robust failure detection systems". In: *IEEE Transactions on Automatic Control* 29.7, pp. 603–614. ISSN: 0018-9286. DOI: 10.1109/TAC.1984.1103593. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1103593.

Console, Luca, Claudia Picardi, and Daniele Theseider Dupré (2007). "A framework for decentralized qualitative model-based diagnosis". In: *IJCAI International Joint Conference on Artificial Intelligence*, pp. 286–291. ISSN: 10450823.

Dastani, Mehdi (2006). "3APL Platform User Guide". In:

Declerck, P. and Marcel Staroswiecki (1991). "Characterisation of the canonical components of a structural graph for fault detection in large scale industrial plants". In: *Proceedings of ECC'91*. Grenoble, France, pp. 298–303.

Duatis, Jordi, Cecilio Angulo, and Vicenç Puig (2014). "FAST: A fault analysis software tool". In: *IEEE Conference on Control Applications (CCA)*. IEEE, pp. 376–381. ISBN: 978-1-4799-7409-2. DOI: 10.1109/CCA.2014.6981375. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6981375.

— (2015). "The Software Architecture of FAST : An Agent-based FDI Tool". In: *9th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes.*

Duatis, Jordi et al. (2008). *Robust Intelligent Systems*. Ed. by Alfons Schuster. London: Springer London, pp. 1–21. ISBN: 978-1-84800-260-9. DOI: 10.1007/978-1-84800-261-6. URL: http://eprints.pascal-network.org/archive/00004534/http://link.springer.com/10.1007/978-1-84800-261-6.

Eduard Muntaner, Esteve del Acebo, and Josep Lluís de la Rosa (2005). "Rescatando Agent-0. Una aproximación moderna a la Programación Orientada a Agentes". In: *I Congreso Español de Informática*.

Ferrari, R (2008). "Distributed fault detection and isolation of large-scale nonlinear systems: an adaptive approximation approach". In: *Openstarts.Units.It*. URL: http://www.openstarts.units.it/dspace/handle/10077/3118.

Frank, Paul M. (1994). "Enhancement of robustness in observer-based fault detection". In: *International Journal of Control* 59.4, pp. 955–981. ISSN: 0020-7179. DOI: 10.1080/00207179408923112. URL: http://www.tandfonline.com/doi/abs/10.1080/00207179408923112.

Gertler, Janos (1991). "Analytical redundancy methods in fault detection and isolation". In: *Proceedings of IFAC/IAMCS symposium on safe process*. Vol. 1. Baden-Baden, Germany, pp. 9–22.

— (1998). *Fault detection and diagnosis in engineering systems*. New York: Marcel Dekker, Inc. ISBN: 0-8247-9427-3.

Isermann, Rolf (1997). "Supervision, fault-detection and fault-diagnosis methods - An introduction". In: *Control Engineering Practice* 5.5, pp. 639–652. ISSN: 09670661. DOI: 10.1016/S0967-0661(97)00046-4.

— (2005). "Model-based fault-detection and diagnosis - Status and applications". In: *Annual Reviews in Control* 29.1, pp. 71–85. ISSN: 13675788. DOI: 10.1016/j.arcontrol.2004.12.002.

— (2006). *Fault-diagnosis systems: An introduction from fault detection to fault tolerance*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–475. ISBN: 3540241124. DOI: 10.1007/3-540-30368-5. URL: http://link.springer.com/10.1007/3-540-30368-5.

Izadi-Zamanabadi, Roozbeh (1999). "Fault-tolerant Supervisory Control : System Analysis and Logic Design". PhD thesis. ISBN: 8790664027. URL: http://vbn.aau.dk/en/publications/faulttolerant-supervisory-control(b8d54e50-0033-11da-b4d5-000ea68e967b).html.

Izadi-Zamanabadi, Roozbeh and Marcel Staroswiecki (2000). "A structural analysis method formulation for fault-tolerant control system design". In: *39th Conference on Decision and Control*, pp. 4901–4902. ISSN: 0191-2216. DOI: 10.1109/CDC.2001.

914707. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=914707.

Julian, V et al. (2002). "SIMBA : an Approach for Real-Time Multi-Agent Systems Social Real-Time Domains". In: *In Proceedings of V Conferencia Catalana d'Intel.ligència Artificial*. Castell: Springer-Verlag. DOI: 10.1.1.99.775.

Kleer, Johan de, Alan K. Mackworth, and Raymond Reiter (1992). "Characterizing diagnoses and systems". In: *Artificial Intelligence* 56.c, pp. 197–222. ISSN: 00043702. DOI: 10.1016/0004-3702(92)90027-U.

Köppen-seliger, Birgit, Steven X Ding, and Paul M. Frank (2002). "MAGIC – IFATIS: EC-Research Projects Birgit Köppen-Seliger, Steven X. Ding and Paul M. Frank". In: *Interface*.

Laird, John E. and Clare Bates Congdon (2006). "The Soar User's Manual - Version 8.6.3". In: URL: http://ai.eecs.umich.edu/soar/sitemaker/docs/manuals/Soar8Manual.pdf.

Lasseur, Christophe et al. (2004). "Life Test Validation of Life Support Hardware in CONCORDIA Antarctic base". In: DOI: 10.4271/2004-01-2352. URL: http://papers.sae.org/2004-01-2352/.

Lehman, Jf (1996). "A gentle introduction to Soar, an architecture for human cognition". In: *Invitation to cognitive ...*

Llanos Rodríguez, Alejandro David (2008). "Time Misalignments in Fault Detection and Diagnosis". PhD Thesis. Universitat de Girona. ISBN: 9788469189009.

Maquin, D and V Cocquempot (1997). "Generation of analytical redundancy relations for FDI purposes". In: *...on Diagnostics for ...* URL: http://hal.archives-ouvertes.fr/docs/00/31/12/54/PDF/Sdemped.pdf.

Merzouki, Rochdi et al. (2013). *Intelligent Mechatronic Systems*. ISBN: 978-1-4471-4627-8. DOI: 10.1007/978-1-4471-4628-5. URL: http://link.springer.com/10.1007/978-1-4471-4628-5.

Michael Rectenwald (2002). *RETSINA Developers Guide*. 1. Carnegie Mellon University, pp. 1–5.

Ould Bouamama, Belkacem et al. (2001). "Diagnosis of a Two-Tank System". In: *Intern Report of CHEM-project* USTL, Lille, France.

Petti, Thomas F., Jim Klein, and Prasad S. Dhurjati (1990). "Diagnostic model processor: Using deep knowledge for process fault diagnosis". In: *AIChE Journal* 36.4, pp. 565–575. ISSN: 0001-1541. DOI: 10.1002/aic.690360408. URL: http://doi.wiley.com/10.1002/aic.690360408.

Ploix, S., S. Gentil, and S. Lesecq (2003). "Fault Detection, Supervision and Safety of Technical Processes Isolation Decision for Multi-Agent Based Diagnostic System". In: *SafeProcess 2003*.

Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf (2005). "A goal deliberation strategy for BDI agent systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3550 LNAI, pp. 82–93. ISSN: 03029743. DOI: 10.1007/11550648_8.

Puig, Vicenç and Carlos Ocampo-Martínez (2015). "Decentralised fault diagnosis of large-scale systems : Application to water transport networks". In: *26th International Workshop on Principles of Diagnosis*. Paris.

Puig, Vicenç et al. (2000). "Model Based Fault Diagnosis of Dynamic Processes : comparing FDI and DX methodologies". In:

Pulido, B. and C.A. Gonzalez (2004). "Possible Conflicts: A Compilation Technique for Consistency-Based Diagnosis". In: *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)* 34.5, pp. 2192–2206. ISSN: 1083-4419. DOI: 10.1109/TSMCB.

2004.835007. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1335515.

Reiter, Raymond (1987). "A theory of diagnosis from first principles". In: *Artificial Intelligence* 32.1, pp. 57–95. ISSN: 00043702. DOI: 10.1016/0004-3702(87)90062-2. URL: http://linkinghub.elsevier.com/retrieve/pii/0004370287900622.

Samantaray, Arun Kumar (2004). "Component-Based Modelling of Thermofluid Systems for Sensor Placement and Fault Detection". In: *Simulation* 80.7-8, pp. 381–398. ISSN: 0037-5497. DOI: 10.1177/0037549704046339.

Shoham, Y (1993). "Agent-oriented programming". In: *Artificial Intelligence* 60.1, pp. 51–92. ISSN: 00043702. DOI: 10.1016/0004-3702(93)90034-9.

Staroswiecki, Marcel, S. Attouche, and M. L. Assas (1999). "A graphic approach for reconfigurability analysis". In: *DX'99*.

Staroswiecki, Marcel, J. P. Cassar, and P. Declerck (2000). "A Structural Framework for the Design of FDI System in Large Scale Industrial Plants". In: *Issues of Fault Diagnosis for Dynamic Systems*. London: Springer London, pp. 245–283. DOI: 10.1007/978-1-4471-3644-6_9. URL: http://link.springer.com/10.1007/978-1-4471-3644-6_9.

Staroswiecki, Marcel and P. Declerck (1989). "Analytical Redundancy in Non-linear Interconnected Systems by Means of Structural Analysis". In: *IFAC'89*, Vol. 2 (1989), pp. 23–27.

Wooldridge, Michael John (1992). "The logical modelling of computational multi-agent systems". PhD Thesis. University of Manchester, p. 163.

— (2002). *An Introduction to Agent Systems*. John Wiley & Sons, Ltd. ISBN: 0-471-49691-X.

Young, Bryan and Ricardo Trindade (2013). *JeasyOpc*. URL: http://sourceforge.net/projects/jeasyopc/ (visited on 10/10/2015).