A Fault Detection and Isolation Software Framework for Repeatable and Comparable Experimentation

Francisco Serdio[1] and Edwin Lughofer[2]

[1,2] *Department of Knowledge-Based Mathematical Systems, Johannes Kepler University Linz, Austria*
francisco.serdio@jku.at
edwin.lughofer@jku.at

ABSTRACT

There is an extensive literature available about condition monitoring relying on multi-dimensional data-driven system models and mappings, including proposal of new methods and algorithms, comparison of state-of-the-art methods, and state-of-the-art revisions. But, when practitioners start to implement their own software to carry out their research, there is a lack of articles in the literature with detailed documentation about how to design a framework for repeatable and comparable experimentation. We propose a design for repeatable and comparable experimentation on the field of Data-Driven Residual-Based Fault Detection and Isolation. The framework has already been used for several experiments, with successful results, eliciting features such as (i) decreasing of developing times, (ii) facilitating of configuration management, and (iii) facilitating of collection and comparison of results.

1. INTRODUCTION

Fault diagnosis plays a central part within modern industrial systems in order to assure condition monitoring and quality control with high performance capabilities (Iserman, 2011) (Korbicz, Koscielny, Kowalczuk, & Cholewa, 2004). Several goals are pursued by the development and installation of such components. While some goals are product-related, placing the focus on the assurance of high quality items/parts at the end of the production chain (Eitzinger et al., 2010), some others are production-related, mainly aiming for minimal operational downtime due to maintenance, degradation or failures inside the system (Palade & Bocaniala, 2010) (Chiang, Russell, & Braatz, 2001). In the ideal case, a zero-defect strategy is pursued in order to exclude any bad production parts, thus saving costs and time-intensive posteriori checks. Human-related factors are vital as well especially within active human-machine interaction scenarios (Lughofer et al., 2009), as operators could be injured when a system suffers from any malfunction not properly addressed (Angelov, Giglio, Guardiola, Lughofer, & Luján, 2006).

Francisco Serdio et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 United States License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Fault diagnostics thereby consists of several steps ranging from Fault Detection (FD) (Chiang et al., 2001) through Fault Isolation (FI – location of the fault) (Gorinevsky, 2011) to fault identification (elicitation of intensity and type of faults) (Mehranbod, Soroush, & Panajpornpon, 2005) as well as finally to fault reasoning (identifying the root cause of a fault) (Wilson, Larry, & Anderson, 1993) and fault correction (automatic self-healing capacities of the system). Intuitively, it is clear that the performance of all these components heavily relies on the FD performance, as only upon the correct detection of faults the remaining steps can be triggered. Thus, the main intention is usually to maximize the fault detection rates. Furthermore, FI plays an important role especially in large-scale production systems (with different stages) (Cohen, Avrahami-Bakish, Last, Kandel, & Kipersztok, 2008) or multi-sensor networks (Khaleghi, Khamis, Karray, & Razavi, 2013) where the manual search for the fault location is very time-intensive and often too slow for omitting severe failure which could be even dangerous for operators (e.g. a leakage in a pipe for emission gases) (Angelov et al., 2006).

There is a vast literature of Fault Detection and Isolation (FDI) methods with the usage of data-driven (statistical) system models, typically explaining the relationships between sensors and measurement channels in form of high-dimensional causal mappings —comprehensive works can be found in (Wang & Gao, 2006) or (Montgomery, 2008); data-driven models have the advantage that they can be fully automatically extracted and thus do not require long development phases as is the case of specific signal (frequency-based) analysis methods (Pichler et al., 2016) or observer-based design based on physical models (Chen & Patton, 1999) (Korbicz et al., 2004). Also our recent works address this way of performing FDI, which are based on an on-line analysis of residual signals extracted from the system models and which have been successfully applied to several industrial use cases —for details see (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014.a)

(Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014b) (Serdio et al., 2015).

In this paper, we present the ***detailed software design*** of our FDI framework for repeatable, comparable and fully automatic experimentation. In cases when the number of sensors/measurement channels becomes huge and many causal relations between channels exist, an appropriate handling of the complex SysId network desires specific requirements regarding architecture and software design. Such a careful design would establish a framework for testing and comparing different FDI strategies, while minimizing the programming efforts, and reducing the bug rates.

To our best knowledge and upon our detailed state-of-the-art researches, we found only one framework of similar nature as proposed by other authors in (Feldman et al., 2010). The greatest difference among this work and ours are:

1. Our framework starts from an absolute lack of knowledge of the system topology, therefore providing interfaces for plugin SysId methods to identify a system network, whereas the work in (Feldman et al., 2010) stars from a known system topology and provides a graph-like representation to express this matter.

2. The work in (Feldman et al., 2010) has a wider goal than ours, as it reveals agnostic regarding the diagnostic method used, whereas ours is Data-Driven Residual-Based specific.

3. The work in (Feldman et al., 2010) includes big efforts on defining metrics of several types, including detection, isolation, computational and system metrics. Such efforts must be schedule for future work to improve our framework, as they reveal cornerstone.

The paper is thus structured in the following way: Section 2 describes the previous knowledge needed for a better understanding of sequel sections, Section 3 enumerates the framework requirements considered, Section 4 describes the detailed design of the FDI framework proposed, focusing on the description of the business model entities and on some services and interfaces, displaying the framework capabilities, Section 5 summarizes the cases of study where the framework has been used, and Section 6 concludes the paper.

2. PREVIOUS KNOWLEDGE

The framework was developed following the Domain Driven Design paradigm. Being familiar with it is recommended, in order to have a better understanding. It is also recommended to be familiar with Object Oriented (OO) programming (Eckel, 2000), as well as with software design patterns (Gamma, Helm, Johnson, & Vlissides, 1995). The following Section 2.1 introduces the Domain Driven Design (DDD) paradigm.

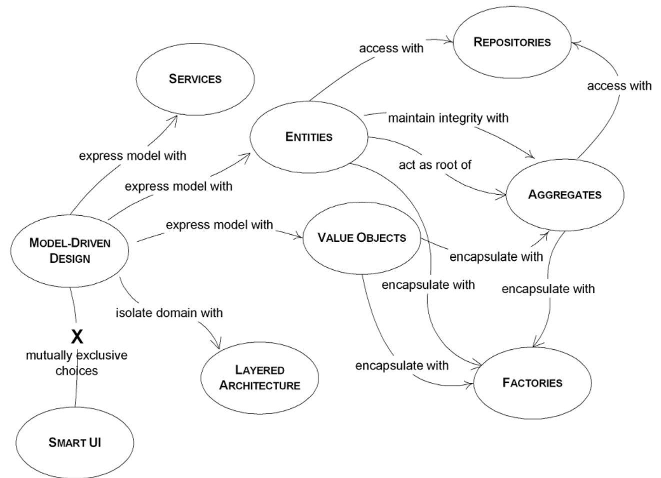

Figure 1. Domain Driven Design explained graphically, reprinted from (Evans, 2006).

2.1. Domain Driven Design

Domain Driven Design (DDD) is a paradigm for software development which centers the design on the business domain. Because it is deeply driven by the domain, its main goal is to reflect the business concepts with the greatest possible accuracy (Evans, 2006). The following explanations about DDD are extracted from (Evans, 2004) and (Evans, 2006), cornerstone readings to understand the paradigm.

Entities are objects which are mainly characterized by an identity. The identity remains the same throughout the states of the software, therefore making possible to track the entities. Entities in our design are concepts such SCENARIO, TRAINFILE, TESTFILE, TRAININGBLOCK, METHOD, METHODTRAINED, EVALUATION, etc.

Value objects are objects describing a certain domain aspect, but without an identity. The main interesting aspect of a value object is how we are interested on its attributes, and not on which object it is (no identity). Value objects in our design are concepts such TRUEPOSITIVERATE, FALSEPOSITIVERATE, etc.

Aggregates define object ownership and boundaries. It is actually a pattern, which groups associated entities and value objects together, so they are considered as one unit from the point of view of data changes and consistency. Thus, the aggregate and all the entities and value objects inside its boundaries are only accessible through a root entity, which is responsible for the integrity of all the aggregate as a whole.

Repositories are the components responsible for encapsulating the logic needed to obtain object references, either to entities or to value objects. They are therefore dealing with the persistence infrastructure (databases, web services, files, etc.), so the domain model is decoupled from it. The repositories must contain the details to access the infrastructure while

keeping a single interface. The main interface of a repository is intended for querying objects. When the repositories access to different types of infrastructure, it is plausible to consider using an strategy (Gamma et al., 1995).

Services are responsible for actions which do not clearly belong to an object. In general they represent important actions or behaviors in the domain. In our domain, a service is for instance (i) the training of a particular method for a concrete scenario, system identification and normalization, (ii) the creation of a set of artificial faults for a given scenario, (iii) the run of a FDI experiment given a method, a set of faults, and an FDI method, etc. Generally speaking, the action or behavior (concept) encapsulated by a service uses several entities and value objects, becoming a connection point for all of them.

3. FRAMEWORK REQUIREMENTS

The following were the gathered requirements, which drove the framework design.

R1 - Perform quality control on processes where only historical data is available, and expert knowledge is not available.

R2 - Allow different normalization actions for pre-processing the available process data.

R3 - Allow different SysId methods for eliciting the system network hidden behind the data.

R4 - Be agnostic about the methods used as residual generators, so any Data-Driven residual generator would suit.

R5 - Provide a unified way for inserting different artificial faults (abrupt and incipient) in test data.

R6 - Provide a unified way for testing different FDI strategies.

R7 - Provide a unified way for repeating the same test scenarios using different FDI strategies.

R8 - Provide a unified way for comparing the results obtained from the same test scenarios on different FDI strategies.

R9 - Provide a unified way for comparing the results obtained from different test scenarios on the same FDI strategies.

R10 - Provide a unified way for storing intermediate results, such as (i) system identification networks, (ii) residuals, (iii) fault warnings, (iv) isolation candidate sets.

4. DETAILED DESIGN

This section describes the detailed design of the framework developed, following the DDD paradigm, which arose naturally by creating a model from the FDI concepts whereas the operations of the framework were born as services. The following descriptions focus on the main business entities, and in most of the cases the data model supporting the persistence of the framework is a direct translation from the business entities to (in our choice) a database structure.

Table 1. DDD mapping, showing the aggregate roots and the entities inside the boundaries of the aggregate.

Aggregate root	Entity
Scenario	Scenario
	Channels
	TrainFile
	TestFile
	Real Fault
System Identification	System Identification
	Dependencies
Normalization	Normalization
Training Block	Training Block
	Methods Trained
Methods	Methods
Run	Run
	Fault Intensity
	Custom Fault
	Iteration
	Evaluation
	Residuals
	FPRs
	FPRs by Channel
FD Algorithm	FD Algorithm
	Detections
	FD TPRs
FI Algorithm	FI Algorithm
	Isolations
	FI TPRs

4.1. Scenario

A SCENARIO is a process from where we have historical data we want to model by Data-Driven methods, to later assess our FDI methods and algorithms. Therefore, a scenario aggregates the following entities in its structure:

- One TRAINFILE (it could be several, but would complicate the design), which holds the dataset for training the Data-Driven methods. This dataset is supposed to be fault-free, therefore the ground-truth of the process to model.
- Several TESTFILES, holding the datasets of the process which we will use for testing the trained methods. The test files are the files where we will either (i) have already existing faults or (ii) will inject artificial faults.
- Several CHANNELS (variables). These channels conform the training and test files. To such matter, the business model must support an assertion ensuring that the channels of a given scenario are the same as the channels of its training and test files, thus maintaining the integrity of the data.

4.2. System Identification

A SYSTEMIDENTIFICATION, in Data-Driven modeling, is a process which given a dataset and an specific variable v_i, provides a list of the dataset variables $\{v_j, ..., v_s\}$ which better model v_i, and where $i \neq j \neq s$. In our proposed design, the

list of variables $\{v_j, ..., v_s\}$ are called the DEPENDENCIES of the variable i. Our system identification methods also provide the linear quality per variable, so it is possible to rank the importance of each identified variable, being also possible to establish mechanisms to determine the optimal number of variables to use in a model.

Understanding the aforementioned behavior as an interface, the different system identification processes are suitable to be hidden behind a virtual factory, therefore allowing for different system identification strategies. Both design patterns are explained in detail in (Gamma et al., 1995).

The system identification concludes in a complete SysId network, as depicted in Figure 2, where the nodes are channels (variables) and the arrows between nodes are input/output structures.

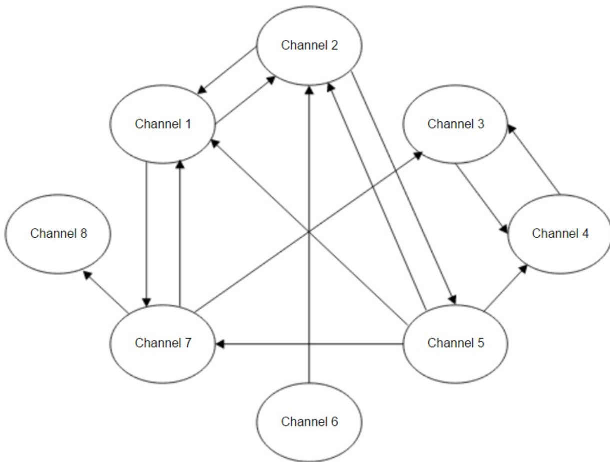

Figure 2. System identification network example including eight channels (sensors), where the arrows indicate the input/output structure of the various models for all channels (each channel used as target once).

In that example, we can elicit several different cases:

- 'Channel 2' has a three dimensional model, with inputs {'Channel 1', 'Channel 5', 'Channel 6'}
- 'Channel 8' is redundant with 'Channel 7', as it has a one dimensional model with inputs {'Channel 7'}
- 'Channel 6' does not have a model, meaning it cannot be explained by the other channels.

4.3. Normalization

A NORMALIZATION represents a collection of actions used to pre-process raw data. Among the different possible actions, the most usual ones are scaling the data to a certain range, mean centering the data, stretching variances to the unit, and denoising. As with the system identification, the normalization is suitable to be hidden behind a virtual factory (Gamma et al., 1995), therefore allowing for different strategies (single actions or combinations of them). When this is over-engineered, it is possible to consider storing the datasets normalized according to the different strategies. There are three important remarks with normalization actions:

- Some actions, such as scaling the data, must be extracted from the training data. There are real cases where the test data is out of the ranges of the training data. The action applied to this data could magnify even more the range disparity. This out-of-range situation could be a first indicator about wrong test data. Reasons are manifold, for instance (i) training and test data were recorded at different process stages, (ii) the data was collected after changes in the process, such as maintenance, repairs, etc.
- Even when actions such scaling, mean centering or stretching variances only affect the quality of the trained methods in case of scale-variant methods (methods affected by the scale of the data), it must be taken into account how the residuals of scale-invariant methods would be affected when misleading or using different factors for these actions during training and testing.
- In some cases, the actions to apply could be extracted from the online streaming test data. This is a particular case, used with, for instance, evolving systems. We do not cover this situation in our framework design.

4.4. Training Block

A TRAININGBLOCK is an important entity regarding the training of Data-Driven methods, as it relates the trained methods (proxies to models) with the scenario and channels they were trained (created) for, using a particular 3-tuple of {normalization, system identification, method}. Thus, the business model allows to browse from a given training block to the concrete method trained associated to a concrete channel of a scenario. The following is the minimal interface proposed for the Trained Block repository:

Interface 1 Proposed interface for the training block repository

1: Input id: the id of the training block
2: Return: the training block with id id
3: **function** SELECTBY_ID(id)
4: Input scn: the scenario of the training block(s)
5: Return: the training block(s) of the scenario scn
6: **function** SELECTBY_SCN(scn)
7: Input scn: the scenario of the training block
8: Input $sysId$: the system identification of the training block
9: Input nrm: the normalization of the training block
10: Return: the training block of the 3-tuple $\{scn, sysId, nrm\}$
11: **function** SELECTBY_SCN_SYSID_NRM(scn, sysId, nrm)

4.5. Method

A METHOD is an entity to discern among the different method types. Thus, each type of method for testing would be a separate method entity. Methods are, for instance, linear regression, fuzzy systems, neural networks, etc.

4.6. Method Trained and IModel

A METHOD TRAINED is a wrapper to a model, making it accessible to the business domain. Hiding the model behind an interface as proposed in Interface 2, the framework allows to perform several operations with exchangeable models. Thus, each model implementing the interface IMODEL could be exchanged in operations such as:

1. Training strategies, like Cross Validation (CV).
2. Assessment of model performance and quality.
3. Estimation of online data.
4. Generation of residuals.
5. FDI algorithms.

Algorithm 1 depicts a generic CV implementation, which profits from the abstraction given by the interface.

Interface 2 Proposed interface IMODEL, to be implemented by the different model types

1: Input $params$: variable list of parameters
2: Constructor for an empty model parameterized by $params$
3: **function** MODEL(params)
4: Input $inputs$: matrix of inputs of the model
5: Input $targets$: vector of targets of the model
6: Creates a model given the $inputs$ and $targets$
7: **function** CREATE_MODEL(inputs, targets)
8: Input $inputs$: matrix of inputs to evaluate
9: Input $targets$: vector of targets to evaluate
10: Return $output$: the evaluation given by the model
11: **function** EVALUATE(inputs, targets)
12: Input $inputs$: matrix of inputs to assess the performance
13: Input $targets$: vector of targets to assess the performance
14: Return $performance$: the performance of the model
15: **function** CHECK_PERFORMANCE(inputs, targets)

4.7. Run

A RUN represent an experiment. The entity aggregates the following entities in its structure:

- Several FAULTINTENSITIES, intended for testing the sensitivity of the methods to the magnitude of the faults occurring inside the system. Concerning this design, our proposed framework includes a component able to introduce faults on the fly, given a test dataset, a channel, a custom fault (fault description) and a fault intensity –see more detail in Section 4.9 and in Algorithm 2.
- Several ITERATIONS. An iteration is a repetition of the experiment, with a different set of faults. Thus, the faults are introduced in the test data whereas being associated to a particular iteration –see Section 4.8.
- Several CUSTOMFAULTS, associated to the iterations, as described in Section 4.9.

4.8. Iteration

As already introduced, an ITERATION is a repetition of a run, with a different set of custom faults. Therefore, each iteration must test its set of faults for all the fault intensities of the run the iteration belongs to. This behavior is suitable to be implemented as a service of the business model, and would compute the indicators for the statistics about the FDI performance of the methods, i.e. how much faults were properly detected and properly isolated. One more time, this behavior is understood as a service –recall Section 2.1.

4.9. Custom Fault

A CUSTOMFAULT is an entity describing a fault. Our design includes two main categories for faults: {*abrupt*, *incipient*}. These faults are introduced on the fly, by a FAULTBUILDER component, as described in Algorithm 2. The incipient faults (drifts) are parameterized by a slope, allowing for different shapes (logarithmic, constant or exponential), reflecting different possible malfunctions. We didn't consider combinations of faults into a single pattern, as the aforementioned fault types are the most common ones (Luo, 2006), and the most studied along the literature as well (Zhang, Polycarpou, & Parisini, 2000) (Zhang, Polycarpou, & Parisini, 2002) (Parlangeli, Pacella, & Corradini, 2007) (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014.a) (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014b) (Serdio et al., 2015).

4.10. Real Fault

A REALFAULT is a fault which is contained in the recorded data of a test file. There are several remarks about real faults, and mixing them with custom faults, in our design:

- Generally speaking, unless the opposite is properly documented, we do not know the ground-truth of the data where a real fault is happening.
- When the ground-truth data where a real fault occurs is unknown, the real fault is not reproducible at different fault intensities.
- When the ground-truth data where a real fault occurs is known, these authors encourage to model that fault as a custom fault, therefore making possible to reproduce it at different fault intensities, thus allowing to determine at what intensity the fault is properly detected and isolated.
- When performing a run with custom faults in a scenario in which test files also have real faults, the regions where real faults occur must be considered, such that detections happening there are not counted as False Positive (FP).
- These authors discourage mixing real faults and custom faults in a run.

4.11. Evaluations

An EVALUATION is the result of applying a method trained (the model behind it) to a test file. Given a method trained, the business model allows to identify the dependencies used for training it (see Figure 3), and by using these dependencies,

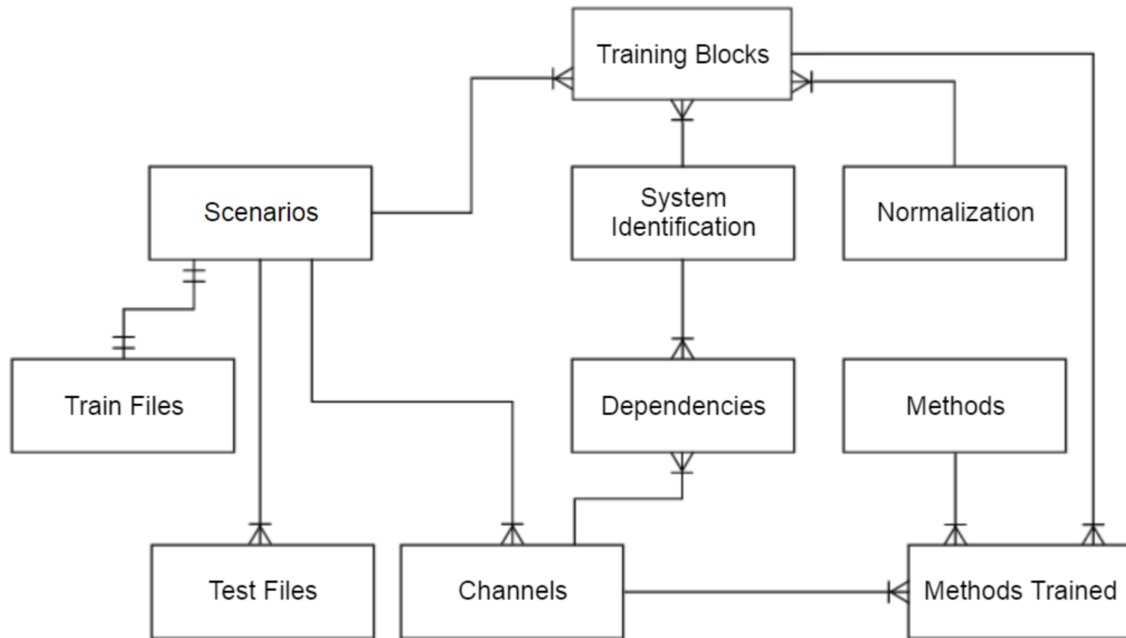

Figure 3. Design of SCENARIOS and TRAININGBLOCKS. A scenario is composed of one training file, and several test files, all containing the same set of channels. The training blocks are the joint entities keeping together the methods trained for the channels of a scenario, given a concrete pair of {system identification, normalization} strategies.

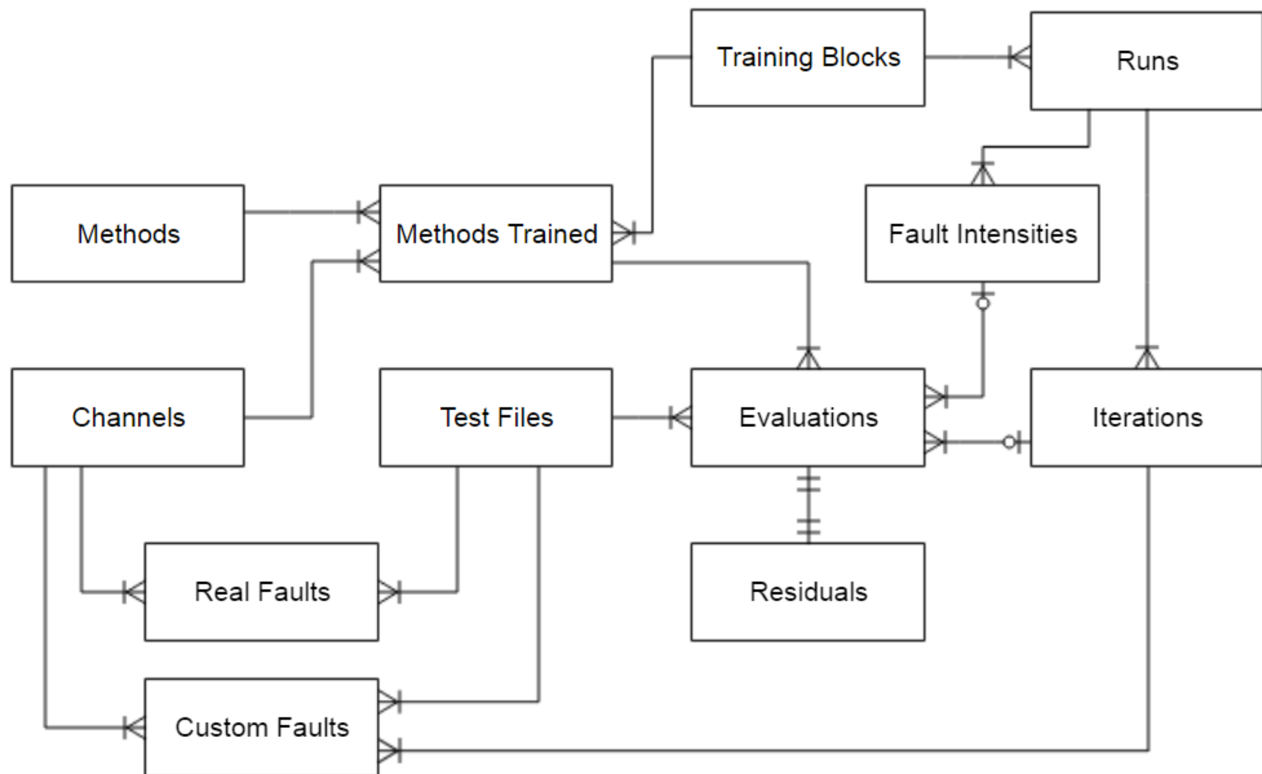

Figure 4. Design of the RUNS of an FDI experiment, where it is seen how a run in composed of several iterations, and how each iteration contains its own set of custom faults, which are reproducible in all the fault intensities of the run. The methods trained are, later on, evaluated on the test files containing the custom faults, computing the residuals which serve as the starting point for the Residual-Based FDI algorithms to be tested.

Algorithm 1 Generic Cross Validation algorithm, training models which implement the IMODEL interface

```
1:  Input inputs: matrix of inputs for training the model
2:  Input targets: vector of targets for training the model
3:  Input model_params: parameters of the model
4:  Return model_final: the trained model
5:  Return perf_final: the model performance on the whole data
6:  Return perf_training: the model performance on the training folders
7:  Return perf_testing: the model performance on the test folders
8:  function CROSS_VALIDATION(inputs, targets, model_params)
9:      folders ← SPLIT(inputs, targets)                    ▷ Split is a function to create the CV folders
10:     for all folder f_i in folders do
11:         in_training_i ← INPUTS_TRAINING(f_i)             ▷ A CV folder is divided in training and test
12:         in_testing_i ← INPUTS_TESTING(f_i)
13:         out_training_i ← TARGETS_TRAINING(f_i)
14:         out_testing_i ← TARGETS_TESTING(f_i)
15:         model_i ← MODEL(model_params)                    ▷ Parameterized empty model
16:         CREATE_MODEL(model_i, in_training_i, out_training_i)   ▷ Model the training data of the CV folder
17:         perf_training(i) ← CHECK_PERFORMANCE(model_i, in_training_i, out_training_i)   ▷ Performance assessment
18:         perf_testing(i) ← CHECK_PERFORMANCE(model_i, in_testing_i, out_testing_i)
19:     end for
20:     model_final ← MODEL(model_params)                    ▷ Final model
21:     CREATE_MODEL(model_final, inputs, target)
22:     perf_final ← CHECK_PERFORMANCE(model_final, inputs, target)
23:     return model_final, perf_final, perf_training, perf_testing
24: end function
```

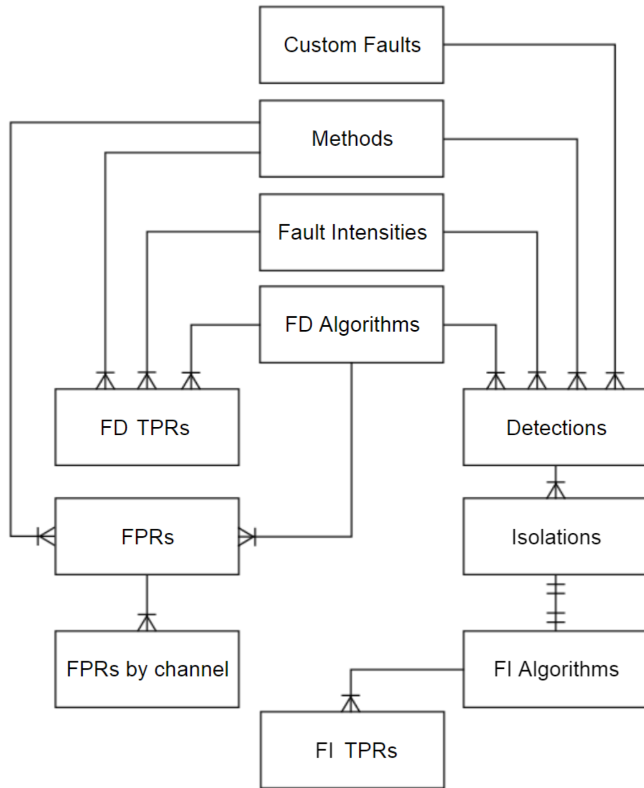

Figure 5. Design of FD performance rates.

it is possible to extract (from the test file) the inputs to the model and the target to estimate.

There are two types of evaluations:

- Evaluations not linked to a pair {iteration, fault intensity}. These evaluations are performed on the test files without custom faults built-in. These are the evaluations used to calculate the FP rates, as they represent the behavior of the running FDI system during the normal operation, i.e. when no faults occur.
- Evaluations linked to a pair {iteration, fault intensity}. These evaluations are performed on the test files with custom faults built-in. These are the evaluations used to calculate the True Positive Rates (TPRs), as they represent the behavior of the running FDI system during the operation with faults.

4.12. Residuals

A RESIDUAL is directly linked to an evaluation by a 1-1 relation, so the residual could be an attribute of the evaluation it belongs to. Our design explicitly separate both entities because of performance, as the residuals of an evaluation must, at least, hold one signal as large as the channel the residuals were computed for. Therefore, when separated, it would be possible to retrieve one or several evaluations from the repositories without uploading into memory the whole residual signals calculated. The residual entity is also suitable to store (i) the estimation of the target from where it was extracted, to avoid browsing the business model to look for it, (ii) some metrics about the residual signal, such are error measures of the estimation, correlation measures between the residuals and the target, etc. Departing from the residuals, practitioners

Algorithm 2 Introduction of custom artificial faults on the fly

```
1:  Input dataset: the dataset to introduce the fault in
2:  Input channel_name: the name of the channel where the fault must be inserted
3:  Input custom_fault: the custom fault to introduce
4:  Input intensity: the fault intensity
5:  function INSERT_CUSTOM_FAULT(dataset, channel_name, custom_fault, intensity )
6:      channel ← GET_CHANNEL(dataset, channel_name)
7:      start ← GET_START(custom_fault)
8:      end ← GET_END(custom_fault)
9:      fault_type ← FAULT_TYPE(custom_fault)
10:     switch fault_type do
11:         case abrupt
12:             ▷ Introduce the abrupt fault in the channel
13:             channel[start, end] ← channel[start, end] + (channel[start, end] * intensity / 100)
14:         case incipient
15:             ▷ Get the drift shape to modify the channel. Log.: {1/5, 1/4, 1/3, 1/2}. Cons.: {1}. Exp.: {2, 3, 4, 5}
16:             slope ← GET_SLOPE(custom_fault)
17:             fault_size ← (start - end)
18:             step ← (fault_intensity / fault_size)                          ▷ Step of the drift
19:             drift ← ([step : step : intensity]^slope)/(intensity^slope/intensity)   ▷ Increasing intensity of the fault
20:             ▷ Introduce the incipient fault (drift) in the channel
21:             for i_step = 1 → fault_size do
22:                 channel[start + i_step] ← channel[start + i_step] + (channel[start + i_step] * drift[i_step] / 100)
23:             end for
24:     end switch
25: end function
```

can extend the business domain to incorporate, for instance, features extracted by a sliding window after processing the residual signal, so incorporating that line of research.

4.13. Fault Detection Algorithms

A FAULTDETECTIONALGORITHM is an algorithm which analyzes a stream of residuals, requiring (i) to raise an alarm, as soon as possible, when a fault is happening in the system and (ii) to remain silent (do not raise an alarm) when a fault is not happening in the system.

In our proposed design, the FD algorithms satisfy an interface, so as it happened with the models, FD algorithms are exchangeable. The interface is described in Interface 3:

Interface 3 Proposed interface IFaultDetection, to be implemented by the different FD algorithms

```
1: Input params: variable list of parameters
2: Constructor for an FD_Strategy parameterized by params
3: function FD_STRATEGY(params)
4: Input residuals: a stream of residuals
5: Return fd_indicators: a stream of FD indicators
6: function RUN_FD(residuals)
```

4.14. Detections

A DETECTION is an indicator, associated to a custom fault, stating whether a fault has been detected given a combination of {FD algorithm, method, fault intensity}. This is illustrated in Figure 5, which shows all the links among the entities. Thus, given the set of custom faults of a run and considering how each custom fault relates to a detection indicator, it is straightforward to compute the FD capabilities of a particular FD strategy.

4.15. FD True Positive Rates

A FD TRUE POSITIVE RATE (FD TPR) is a percentage in [0-100] indicating how plausible is, for an FD strategy, to detect a fault. The FD TPRs are to be reported for a given combination of {FD algorithm, method, fault intensity}.

4.16. False Positive Rates

A FALSE POSITIVE RATE (FPR) is a percentage in [0-100] indicating how plausible is, for an FD strategy, to raise an alarm indicating a fault when there is not such a fault happening at the system. Similar to FD TPRs, FPRs are reported for a combination of {FD algorithm, method}.

4.17. Fault Isolation Algorithms

A FAULTISOLATIONALGORITHM is an algorithm which, given a FD indicator, analyzes the state of the system requiring (i) to provide a candidate list of variables responsible for the fault, and (ii) to associate a confidence to each variable into the candidate list. Following the same line of reasoning as with FD algorithms, FI algorithms must satisfy an interface, in order to become exchangeable.

4.18. Isolations

An ISOLATION is an indicator, associated to a detection, stating whether a fault has been isolated given a combination of {FI algorithm, method, fault intensity}. The combination {FI algorithm, method, fault intensity}, in our design, is indirectly associated to the isolation by the (previous) detection. Please recall Figure 5 for a better understanding.

4.19. FI True Positive Rates

A FI TRUE POSITIVE RATE (FI TPR) is a percentage in [0-100] indicating how plausible is, for an FI strategy, to isolate a fault. As it was the case for detections, the FI TPRs are reported for a combination of {FD algorithm, method, fault intensity}.

5. CASE STUDIES & LESSONS LEARNED

5.1. Case studies tested

We have successfully evaluated the whole framework together with the interplay of all components based on industrial use cases in rolling mill and engine test bench applications, in our previous works in (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014.a), (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014b) (Serdio et al., 2015).

From the case studies tested, we want to highlight that a nice example regarding our claims for an easy usage of the framework with decreasing development times and advantages for configuration management is underlined by our work in (Serdio, Lughofer, Pichler, Buchegger, Pichler, & Efendic, 2014). It tested different filters applied to the residual space, with different configurations, aiming for checking whether it was possible to reduce FPR and increase TPR rates significantly (which was finally the case). Each filter and configuration was developed by creating a different preprocess for a given FD strategy ($\mu + n\sigma$ tracking), and the filter with its configuration was provided by factory. In average, only 5 additional lines of code (plus the factory) were required for each combination of filter+configuration, whereas the processes of (i) residual generation, (ii) test of faults, (iii) computation of FPR and TPR rates and (iv) ROC curves remained untouched.

5.2. Compatibility

The realizations of all the framework components have been implemented in a MATLAB 2010b environment, using for persistence MySql Server 5.5.50. Hence, the framework can be directly used without any modifications under operating systems supporting the combination of both tools, what is mostly true in all recent versions of Windows, Linux and Macintosh.

5.3. Resources

Concerning computational and memory demands (using an Intel Core i7-2600K processor at 3.4GHz, 8 cores, with 12GB of RAM), a bottleneck could be observed during training the large variety of models (e.g., 145 in sum for rolling mills) when using a non-linear model structure and carrying out a detailed cross-validation procedure based on 10-folds.

Such was the case with SparseFIS (Lughofer & Kindermann, 2010), which is a Takagi-Sugeno fuzzy systems automatically extracted from data: it lasted several days up to one week when performing it sequentially for each channel based on a parameter grid over the essential learning parameters. However, the usage of the non-linear model structure was necessary to assure a higher FD performance due to the non-linearities contained in the system to meet the companies goals, and could not be circumvented by linear or quasi linear methods (Serdio, Lughofer, Pichler, Buchegger, & Efendic, 2014.a).

Therefore, parallelized threads for training the models for the different channels helped to reduce the computational burden significantly. This parallelization is straightforward due to the independent treatment given to the channels.

5.4. Lessons Learned

We can not conclude without mentioning some important lessons learned from the design and development of this framework, including the open lines where the framework still needs efforts and future extensions.

One important extension line is the inclusion of a mechanism to simulate a data acquisition component, so the framework could provide functionality for dealing with (i) missing data, (ii) noisy data and (iii) data recorded at different rates, coming from different subsystems. At its current state, the framework does not provide a mechanism allowing this pre-processing to adequate the data, so it must be carried out outside the framework. Once carried out, the resulting datasets can be used to create an Scenario and test the FDI algorithms, but functionality to integrate these data-adquisition steps would considerably improve the framework.

As it was stated in Section 4.9, the most common fault types are {*abrupt*, *incipient*} (Luo, 2006), and there are the ones the framework can introduce on-the-fly. Future versions must also include intermittent faults and multiple faults (faults overlapped on time), therefore expanding the testing casuistry.

Another important extension line is to incorporate other measures enriching the set of FDI capabilities assessed inside the framework. When the most important ones are perhaps Fault Detection Time (M_{fd}) and Fault Isolation Time (M_{fi}), CPU Load (M_{cpu}) and Memory Load (M_{mem}) are to be considered as well. These and other metrics for future extensions of the framework can be found in (Feldman et al., 2010).

Finally, once the framework provides support for repeatable and comparable FD and FI experimentation, support to test fault severity estimation algorithms seems necessary as well, so the framework could provide full FDI support.

6. CONCLUSION

The paper presents a detailed design of a FDI Software Framework for repeatable and comparable experimentation, which design was guided by its business model following the Domain Driven Design paradigm. The proposed framework was implemented and tested in two real use cases, which related works are already published and available. The article considered software-related aspects, bringing the importance of an architecture to the discussion.

Future work must go in the direction of providing expandability of the framework by the use of public interfaces, considering to make the implementation available for the research community, aiming for the framework maturation.

The main lines for future work are data-acquisition components, inclusion of intermittent and multiple faults and enrichment of the set of (general purpose) FDI measures of the framework.

Finally, a more ambitious extension would consider supporting fault identification, fault severity estimate and fault reasoning algorithms, thus having a full diagnostics system available, which may increase users' attention and understandability of fault alarms. On the other hand, identification and reasoning usually requires significant input from experts (e.g., in form of fault signatures, fault patterns, root-cause rules for fault back-tracing etc.), which would make the framework less generically applicable.

ACKNOWLEDGMENT

The authors acknowledges the support of the Linz Center of Mechatronics (LCM) in the framework of the Austrian COMET-K2 programme. The work-related projects are supported by the Austrian Government and Johannes Kepler University Linz. This publication reflects only the authors views.

REFERENCES

Angelov, P., Giglio, V., Guardiola, C., Lughofer, E., & Luján, J. (2006). An approach to model-based fault detection in industrial measurement systems with application to engine test benches. *Measurement Science and Technology*, *17*(7), 1809–1818.

Chen, J., & Patton, R. (1999). *Robust model-based fault diagnosis for dynamic systems*. Norwell, Massachusetts: Kluwer Academic Publishers.

Chiang, L., Russell, E., & Braatz, R. (2001). *Fault detection and diagnosis in industrial systems*. London Berlin Heidelberg: Springer.

Cohen, L., Avrahami-Bakish, G., Last, M., Kandel, A., & Kipersztok, O. (2008). Real-time data mining of non-stationary data streams from sensor networks. *Information Fusion*, *9*(3), 344–353.

Eckel, B. (2000). *Thinking in c++. volume one: Introduction to standard c++*. Upper Saddle River, New Jersey: Prentice Hall Inc.

Eitzinger, C., Heidl, W., Lughofer, E., Raiser, S., Smith, J., Tahir, M., ... van Brussel, H. (2010). Assessment of the influence of adaptive components in trainable surface inspection systems. *Machine Vision and Applications*, *21*(5), 613–626.

Evans, E. (2004). *Domain-Driven Design: Tackling complexity in the heart of software* (1st ed.). Boston, MA, USA: Pearson Education Inc.

Evans, E. (2006). *Domain Driven Design quickly*. C4Media Inc.

Feldman, A., Kurtoglu, T., Narasimhan, S., Poll, S., Garcia, D., de Kleer, J., ... van Gemund, A. (2010). Empirical evaluation of diagnostic algorithm performance using a generic framework. *International Journal of Prognostics and Health Management*(2), 2153–2648.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. New York, USA: Addison-Wesley.

Gorinevsky, D. (2011). Bayesian fault isolation in multivariate statistical process monitoring. In *Proceedings of the american control conference* (pp. 1963–1968). San Francisco, CA, U.S.A..

Iserman, R. (2011). *Fault-diagnosis applications: Model-based condition monitoring: Actuators, drives, machinery, plants, sensors, and fault-tolerant systems*. Heidelberg Dordrecht: Springer.

Khaleghi, B., Khamis, A., Karray, F. O., & Razavi, S. N. (2013). Multisensor data fusion: A review of the state-of-the-art. *Information Fusion*, *14*(1), 28–44.

Korbicz, J., Koscielny, J., Kowalczuk, Z., & Cholewa, W. (2004). *Fault diagnosis - models, artificial intelligence and applications*. Berlin Heidelberg: Springer Verlag.

Lughofer, E., & Kindermann, S. (2010). SparseFIS: Data-driven learning of fuzzy systems with sparsity constraints. *IEEE Transactions on Fuzzy Systems*, *18*(2), 396–411.

Lughofer, E., Smith, J. E., Caleb-Solly, P., Tahir, M., Eitzinger, C., Sannen, D., & Nuttin, M. (2009). Human-machine interaction issues in quality control based on on-line image classification. *IEEE Transactions on Systems, Man and Cybernetics, part A: Systems and Humans*, *39*(5), 960–971.

Luo, M. (2006). *Data-driven fault detection using trending analysis* (Unpublished doctoral dissertation). Department of Electrical and Computer Engineering, Louisiana State University and Agricultural and Me-

Algorithm 3 Service to perform a Run and compute the Evaluations and Residuals

```
1:  Input run: the run we want to evaluate
2:  Input method: the method we want to test
3:  function PERFORM_BY_RN_MT(run, method )
4:      ▷ The entities are retrieved closer to their use for a better understanding
5:      ▷ The authors are aware that this is not optimal in performance
6:
7:      iterations ← GET_ITERATIONS(run)
8:      for all iteration in iterations do
9:
10:         testFiles ← GET_TESTFILES(run)
11:         for all testFile in testFiles do
12:
13:             faultIntensities ← GET_FAULT_INTENSITIES(run)
14:             for all faultIntensity in faultIntensities do
15:
16:                 customFaults ← GET_CUSTOM_FAULTS(iteration)
17:                 testFile_faults ← BUILD_FAULTS( testFile, customFaults, faultIntensity )
18:
19:                 normalization ← GET_NORMALIZATION(run)
20:                 NORMALIZE(normalization, testFile_faults)
21:
22:                 repositories ← GET_REPOSITORIES(this)
23:                 methodsTrained ← SELECT_MTT_BY_TB_MT(repositories)
24:                 for all methodTrained in methodsTrained do
25:                     channel ← GET_CHANNEL(methodTrained)
26:                     sysId ← GET_SYSID(run)
27:                     dependencies ← GET_DEPENDENCIES(sysId, channel)
28:                     inputs ← GET_INPUTS(testFile_faults, dependencies)
29:                     targets ← GET_TARGET(testFile_faults, channel)
30:
31:                     performance ← CHECK_PERFORMANCE(methodTrained, inputs, targets)
32:                     evaluation ← GET_EVALUATION(performance)
33:                     residuals ← GET_RESIDUALS(performance)
34:
35:                     OPEN_TRANSACTION(repositories)
36:                     INSERT_EVALUATION(repositories, evaluation)
37:                     INSERT_RESIDUALS(repositories, residuals)
38:                     COMMIT_TRANSACTION(repositories)
39:                 end for
40:             end for
41:         end for
42:     end for
43: end function
```


chanical College, Louisiana, LA, USA.

Mehranbod, N., Soroush, M., & Panajpornpon, C. (2005). A methods of sensor fault detection and identification. *Journal of Process Control*, *15*(3), 321–339.

Montgomery, D. (2008). *Introduction to statistical quality control (6th edition)*. John Wiley & Sons.

Palade, V., & Bocaniala, C. (2010). *Computational intelligence in fault diagnosis*. London: Springer.

Parlangeli, G., Pacella, D., & Corradini, M. L. (2007). Fault identification and accommodation for incipient and abrupt faults. In *Proceedings of the 46th ieee conference on decision and control* (pp. 1003–1008). New Orleans, LA, USA.

Pichler, K., Lughofer, E., Pichler, M., Buchegger, T., Klement, E., & Huschenbett, M. (2016). Fault detection in reciprocating compressor valves under varying load conditions. *Mechanical Systems and Signal Processing*, *70–71*, 104–119.

Serdio, F., Lughofer, E., Pichler, K., Buchegger, T., & Efendic, H. (2014.a). Fault detection in multi-sensor networks based on multivariate time-series models and orthogonal transformations. *Information Fusion*, *20*, 272–291.

Serdio, F., Lughofer, E., Pichler, K., Buchegger, T., & Efendic, H. (2014b). Residual-based fault detection using soft computing techniques for condition monitoring at rolling mills. *Information Sciences*, *259*, 304–330.

Serdio, F., Lughofer, E., Pichler, K., Buchegger, T., Pichler, M., & Efendic, H. (2014). Reducing false positives for residual-based on-line fault detection by means of adaptive filters. In *Proceedings of the ieee smc 2014 conference* (pp. 2803–2808). San Diego, CA, U.S.A..

Serdio, F., Lughofer, E., Pichler, K., Pichler, M., Buchegger, T., & Efendic, H. (2015). Fuzzy fault isolation using gradient information and quality criteria from system identification models. *Information Sciences*, *316*, 18–

39.

Wang, L., & Gao, R. (2006). *Condition monitoring and control for intelligent manufacturing*. London, UK: Springer Verlag.

Wilson, F., Larry, D., & Anderson, G. (1993). Root cause analysis: A tool for total quality management. *ASQ Quality Press*, 8–17.

Zhang, X., Polycarpou, M., & Parisini, T. (2000). Abrupt and incipient fault isolation of nonlinear uncertain systems. In *Proceedings of the american control conference* (Vol. 6, pp. 3713–3717). Chicago, IL, USA.

Zhang, X., Polycarpou, M., & Parisini, T. (2002). A robust detection and isolation scheme for abrupt and incipient faults in nonlinear systems. *IEEE Transactions on Automatic Control*, *47*(4), 576–593.

BIOGRAPHIES

Francisco Serdio was born in Asturias, Spain, in 1979. He received the Dipl.-Ing. degree in Computer Science and the MSc. degree in Soft Computing and Intelligent Data Analysis from the University of Oviedo, Asturias, Spain, in 2002 and 2011, respectively. He received the PhD. degree in Engineering Sciences from the Johannes Kepler University, Linz, Austria. He was involved in commercial software development companies from 2003 to 2010, including banking, stock market, CRM, and others. From 2011 on, he got into research, where his main interests include data-driven modeling, fault detection and diagnosis, condition monitoring, prognostics and health management, quality control, fuzzy systems, adaptive learning, multi-objective optimization, genetic algorithms, evolutionary computation, computational intelligence and metaheuristics. He received best paper awards at the IFAC Conference on Manufacturing Modelling, Management and Control, 2013 (800 participants).

Edwin Lughofer received his PhD. degree from the Johannes Kepler University Linz (JKU) in 2005. He is currently Key Researcher with the Fuzzy Logic Laboratorium Linz / Department of Knowledge-Based Mathematical Systems (JKU) in the Softwarepark Hagenberg. During the past 10-12 years, he has participated in several basic and applied research projects on European and national level, with a focus on applications within several topics of Industry 4.0 and FoF (Factories of the Future). In this period, he has published around 140 journal and conference papers in the fields of evolving fuzzy systems, machine learning and vision, data stream mining, active learning, classification and clustering, fault detection and diagnosis, condition monitoring as well as human-machine interaction, including a monograph on Evolving Fuzzy Systems (Springer, Heidelberg) and an edited book on Learning in Non-stationary Environments (Springer, New York). He is associate editor of the international journals IEEE Transactions on Fuzzy Systems (IEEE press), Evolving Systems (Springer), Information Fusion (Elsevier), Soft Computing (Springer) and Complex and Intelligent Systems (Springer), the general chair of the IEEE Conference on Evolving and Adaptive Intelligent Systems (IEEE EAIS) 2014 in Linz, the publication chair of IEEE EAIS 2015 and 2016, and the Area chair of the FUZZ-IEEE 2015 conference in Istanbul. In 2006 he received the best paper award (as sole author) at the International Symposium on Evolving Fuzzy Systems, and in 2013 the best paper award (as co-author) at the IFAC Conference in Manufacturing Modeling, Management and Control Conference (800 participants).