The Fault Library - A New Modelica Library Allowing for the Systematic Simulation of Non-Nominal System Behavior

Julia Gundermann[1] Artem Kolesnikov[1] Morgan Cameron[2] Torsten Blochwitz[1]

[1]ESI ITI GmbH, Dresden, Germany
[2]ESI Group, Aix-en-Provence, France,
{julia.gundermann,artem.kolesnikov,morgan.cameron,torsten.blochwitz}@esi-group.com

Abstract

We introduce a library developed at ESI ITI for modeling faults in physical systems. We outline the motivation of how and why to model faults as well as a description of the library structure. The new library is exemplified in different fields of applications. In addition, we demonstrate a number of complementary tools and techniques for analyzing the results of simulations of faulted models.
Keywords: Fault, FAME, Reliability, Test case generation, Model based diagnosis

1 Introduction

To date, most Modelica libraries consider physical systems in their nominal configuration. There have been limited attempts to extend model coverage towards behavior beyond that. These include the Fault triggering library (van der Linden, 2014), which allows for the insertion of *faults* into models of existing components; and the FAME library (de Kleer et al., 2013). The latter has served as a basis for the development presented in this paper.

The range of applicability of the Fault library we introduce below is broad. We want to illustrate this based on the examples we studied in the scope of our development. They are listed below.

1. Braille printer: We take as a starting the point, a model of a braille printer, which includes mechanical, electrical and magnetic parts. Under normal operation, the force from the magnetic system pushes the needle into the paper. We would like to know, which *faults* in the system might prevent the embossing, leading to a rupture of the paper, prevent the needle from moving back to its initial position, or lead to overheating of the magnetic part.

2. Automotive transmission: The specification for a car contains requirements such as "The car should be able to accelerate from 0 to 100 km/h in 5 Seconds." We want to test the fulfilment of these and other criteria based on the model of a car. Furthermore, we want to identify critical *faults* in the transmission system that lead to a violation of these criteria. On the other hand, one could ask the question whether it is possible to determine the presence of *faults* from some measurable output variables such as velocity or fuel economy, or some other sensor data.

3. Battery package: Battery management systems are used in industry for the protection and operation of Li-Ion battery packs only within safe operating conditions. Because of various requirements the battery management system has to monitor and protect each cell against over-/under-voltage, over-/under-temperature, over-current and other *faults* (Xing et al., 2011). This leads to the necessity of an uniform testing of the battery management system for every type of battery pack. We want to generate *faults* for the battery management system in the battery cell simulator systematically and automatically reuse the model in different battery packs.

4. Feed axis: In view of Industry 4.0, predictive maintenance becomes more feasible in production machinery because of the technical ability to handle large data volumes. Modern machine tools have intelligent drives with control systems e.g. main and feed axis drives, or auxiliary drives (Friedrich et al., 2016). The control system data can be used for *faults* detection and prediction without additional measurement equipment. We would like to know how signal *faults* in the control system of a feed axis might be detected by using measured signals in the feed axis drive.

2 Defining Faults

In the preceding section we spoke about *faults*. In the following we will define more clearly what is meant by this term.

The reasons a physical system can deviate from its designed ("nominal") behavior are manifold. First of all, the manufacturing of products is only exact to a certain point - there are variations and sometimes systematic deviations from the specified system design. Secondly, during the use of a product, its behavior can change. Materials and material pairings are subject to wear, aging, abrupt failures etc. Effects include loss of lubrication, corrosion and hardening of materials.

Throughout this paper the term *fault* refers to any of these processes - i.e. it is a deviation from nominal behavior. It shall be emphasized that we do not care about

the underlying physical processes, let alone its dynamical evolution and/or interaction with other processes. The Modelica models developed for the Fault library simulate only the effect of a process on the system's behavior. For example, mechanical *faults* in joints, gears, shafts, springs or clutches because of breakage or slipping leads to the reduction of transmitted force/torque. The current reduction because of bad or open connections in an electrical system, the mass flow decrease in a hydraulic system because of leakage or obstruction as well as changes of entropy flows in the thermodynamics have related behavior and can be similar structured and analyzed by modeling (Herrmann, 2006).

3 The Fault Library

In the following section the Fault library is described. Its development is based on the FAME library (de Kleer et al., 2013; Minhas et al., 2014). This library is the basis of three fields of applications outlined in Sec. 4. The Fault library consists of a `Basic` package containing elementary type definitions to parametrize *faults*. Beyond that, its structure is oriented according to the structure of composed models in the Modelica language (van der Linden, 2014)

3.1 Type Definition: Fault - Continuous and Discrete

Based on the processes and effects outlined above (cf. Sec. 2) two types of *faults* are defined: continuous and discrete. The library contains appropriate type definitions for both these *fault* types- Continuous.Fault and Discrete.Faults are records, which contain an editable variable, respectively `Real intensity` or `Integer active`. Rather than being fixed parameters, these are dynamic variables which can change their value during simulation. By using a type definition for the *faults*, it is easy to systematically read out and control the *faults* in a model.

Continuous faults parameterize the strength of a gradual effect. Typical examples are wear and aging phenomena which can lead to a gradually different value of a backlash parameter, or friction coefficient, etc. In the `record Continuous.Fault` this gradual change is translated into a normalized `Real` named *intensity*, ranging between zero (nominal) and one (maximal effect). If the *fault* modifies a parameter or variable, the extended `record Continuous.FaultWithFunction` can be used, which allows for the definition of the functional dependency of parameter change, e.g. multiplicative $p = p_0(1 + c_0(\texttt{intensity} * \texttt{scale})^n)$, exponential $p = p_0 \exp\{c_0(\texttt{intensity} * \texttt{scale})^n\}$, or others. Herein, p_0 is the nominal parameter value, and c_0, n are parameters of the function. The `scale` parameter is needed to define a typical magnitude of the *fault* effect, which depends on the specific application, e.g. the mass of the surrounding components, acting forces, etc. For details, see Sec. 3.3.

Discrete faults are either *active* or not, i.e. the *fault* is switched on or off. Typical examples are abrupt phenomena, such as failure of an electronic component, or breaking of a mechanical connection. The `record Discrete.Fault` contains the `Boolean active`, which is `false` in the nominal case, and `true` if the *fault* is active. If the *fault* modifies a parameter or variable, the extended `record Discrete.FaultWithFunction` can be used, which allows for the definition of the functional dependency of parameter change: proportional to p_0 -including setting the prefactor d_0, or as an alternative value - including setting the value p_1.

3.2 Types of Faults in a Model

A Modelica model can consist of components taken from different libraries. These components are connected with each other. *Faults* change the behavior of components (FaultAugmentedModels), alter the transport of quantities between components (ConnectorFaults) or add connections (BridgeFaults). A snapshot of a model which was augmented with components from the Fault library, is shown in Figure 1.

ConnectorFaults can be inserted at (connected) connectors or into existing connections. The first type cuts the connection (in Figure 1, below), it has two connectors, and can be used to model, for example, the breaking off of a mechanical component, or a mechanical obstruction. The second type has only one connection (in Figure 1, at the top), it is added to a connection, and can be used to model, for example an additional friction force/-torque in a mechanical connection, or a leakage in hydraulics. The models of these *faults* consist of a *fault* and a set of equations which depend on the intensity/active(-ity) of this *fault*, and, in general, on a scale.

BridgeFaults are models with two connectors (in the middle of Figure 1). For each domain there exists one BridgeFault. They can be inserted between existing connections, adding further connect-statements, if the *fault* is `active`.

FaultAugmentedModels with parametric *faults* are fault-augmented counterparts to the nominal SimulationX library models. For example, Figure 1 shows a `FaultAugmentedModels.Electricity.Analog.Basic.Resistor`, which is an extension of the `Electricity.Analog.Basic.Resistor`. It contains a *fault*, which changes the resistance parameter `r=Continuous.ChangeParameter(1,fault)` from its default value 1, as defined by `intensity`, `scale`, and `functionType` in the `fault`-record. The Fault triggering library (van der Linden, 2014) models *faults* in a similar way to those contained within FaultAugmentedModels.

DOI 10.3384/ecp18148161

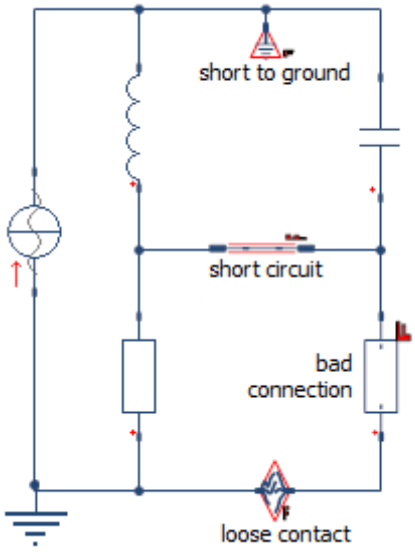

Figure 1. Screenshot of a model of an electrical circuit which has been augmented with faults. One of the resistors is replaced by its `FaultAugmentedModels` counterpart. To the connection between the inductor and the capacitor a `ConnectorFault` modeling a short to ground has been added. The connection between the two resistors has been cut by a `ConnectorFault` modeling a loose contact. An additional switchable connection (BridgeFault) modeling a short circuit between two junctions in circuits of consuming components.

3.3 The Fault Scale

It became clear, especially when modeling the connector or bridge *faults*, but also during the development of the fault-augmented counterparts of the library components, that for most of the *faults* the amplitude of the strongest effect (intensity=1, or active=true) differs depending on the surroundings of the system into which the *fault* is inserted. For example, the frictional torque in a rotational connection which prevents the latter from moving is several orders of magnitude higher in a ship's propeller than in a clockwork mechanism.

In the library this phenomenon is captured with the definition of a positive Real parameter named `scale`, which can be included and defined in the definition of a *fault*, if needed. It should be emphasized that the meaning of the scale differs between different *faults*. Furthermore, scales of the same *faults* in a model can differ in different places - e.g. if the model contains both the ship's propeller and the clockwork mechanism.

For a meaningful effect of the inserted *faults*, it is crucial to find ways to estimate the scale for each *fault*. In the optimal case, one can calculate the scale from the amplitudes of variables in the surroundings of the *fault* as recorded during a single simulation of the nominal behavior. For example, the scale of the translational sticking *fault* is a prefactor to the friction force added to the connection. The latter is defined as F= 1N$\star$`scale`$\star$`intensity` for scale 1. The scale should be chosen to be high enough to ensure that the connection stops moving when the *fault* intensity is equal to one. On the other hand it should be low enough that one is not faced with numerical issues, because the system is stiff or the solver has trouble integrating when the *fault* is switched on during simulation time. In the optimal case the scale value should be set to ensure that the range of the friction force over increasing intensity is such that for low intensities it already has some (recognizable) effect on the connection but does not already prevent it from motion at all (sensitivity to *fault* intensity).

Estimating the scale - even in the seemingly simple case of the friction force from above - is not as straightforward as one might think and is indeed a dedicated research topic. Currently, several methods (reading from base unit force, energy, power, etc.) are under investigation.

4 Fields of Application

Having illustrated the definition of a *fault* and the structure of the Fault library in detail, in the following section we want to give more details about what this can be used for. There are three fields of application we see for the Fault library.

4.1 Reliability

This field is probably the most obvious application. Since any real component has been manufactured, and is subject to wear and aging processes, it can be helpful to estimate the impact of such *faults* at an early development stage. With the Fault library the following questions can be addressed: How does my system behave when *faults* are active? What are the critical *faults* or *fault* combinations with respect to some performance goal? Is it always true that increasing the intensity of a *fault* worsens the performance of the system?

With the Fault library we can use system simulation to answer these questions. Beyond that we have developed another library named *Performance Indicators* to systematically measure whether a system meets or violates predefined criteria (cf. Sec. 6).

The examples "Braille Printer" and "Automotive Transmission" introduced in Section 1 fit into this field of application. For details about the Braille Printer and Automotive Transmission models, as well as the analysis of *fault* states, see Figure 2 and Figure 3.

In general, this analysis can serve as a basis to estimate the system's reliability. In principle, when assigning a probability to each *fault*, one could calculate the overall system failure probability. This could serve as a simulation-based counterpart to graph-based fault tree analysis - which is also possible in SimulationX. Since literature (Bertsche et al., 2009) and customer feedback suggest that these probabilities are in general hard to procure, we have not developed this analysis any further.

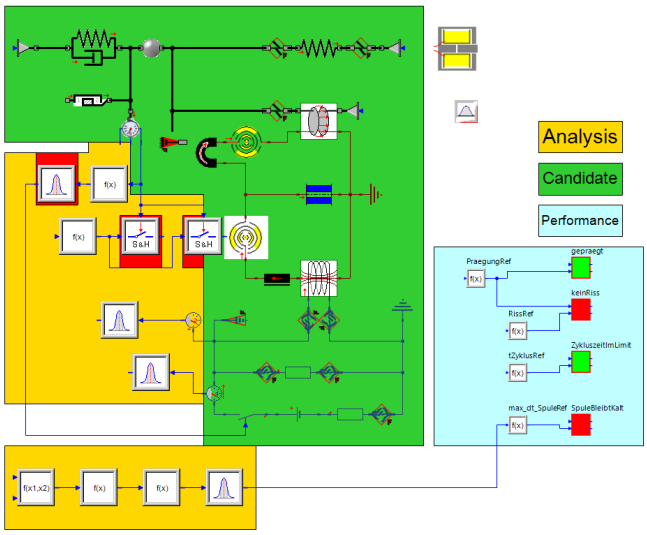

Figure 2. Screenshot of the SimulationX model of a Braille Printer (Kamusella, 2016). The components in the model are categorized - different categories are highlighted by color: green represents the physical components which are subject to faults. Mechanical connector faults (slipping, broken) and faults in the electrical (short to ground, open contact) connections are considered. The yellow area highlights components used for the analysis of the model, which are helpers for the components in the blue area. With the latter the performance of the system is evaluated. From top to bottom the performance indicators (cf. Sec. 6.2) check whether: the paper gets embossed, the paper is not pushed through, the needle is back to its initial position in time, and the magnetic part is not overheated. The fulfillment (green) or violation (red) of a performance criterion in the current simulation are displayed explicitly in the diagram.

4.2 Virtual Testing

The application "virtual testing" refers to model-in-the-loop, software-in-the-loop or hardware-in-the-loop (HiL) tests. In this scenario, the functionality of a controller is investigated through its connection to a model of the physical system. The model or code of the controller has to be able to handle nominal and non-nominal behavior. The latter can manifest itself in a multiplicity of ways - a high amount of modeling/coding effort is devoted to this subject (Plummer, 2007).

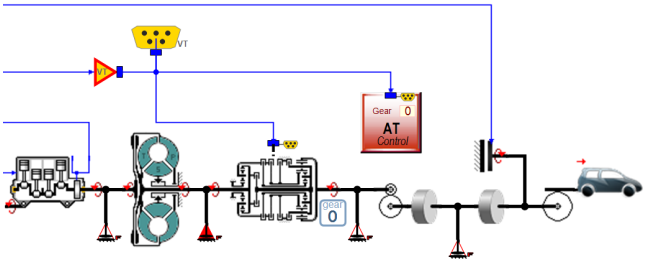

Figure 3. Screenshot of a SimulationX model of an automatic transmission. In the mechanical connections faults adding friction are included. One fault - between converter and transmission - is active (marked in red).

By inserting *faults* from the Fault library into the physical (SimulationX) model of the surrounding system, a large number of test cases for the controller can be generated. The number of *fault* combinations can likewise be very large, hence a semi-automatic algorithm can be used to insert the *faults* (cf. Sec. 5). Since *faults* can arise abruptly during simulation it is necessary that the *fault* intensity/active(-ity) are dynamic variables instead of parameters.

The example "Battery Package" introduced in Section 1 is subject to various *faults* and their combinations. Depending on the surrounding system, it is important to be able to detect the *fault* and behave according to the mechanical, electrical and environmental restrictions for the safe operation of the battery. For details about the Battery Package model see Figure 4. One can easily imagine that the exploration space of *fault* combinations is very large, especially when the option of activating/deactivating *faults* during runtime is included - therefore efficient ways of checking combinations and planning further tests have to be investigated (Tatar and Mauss, 2014; Ruiz et al., 2018).

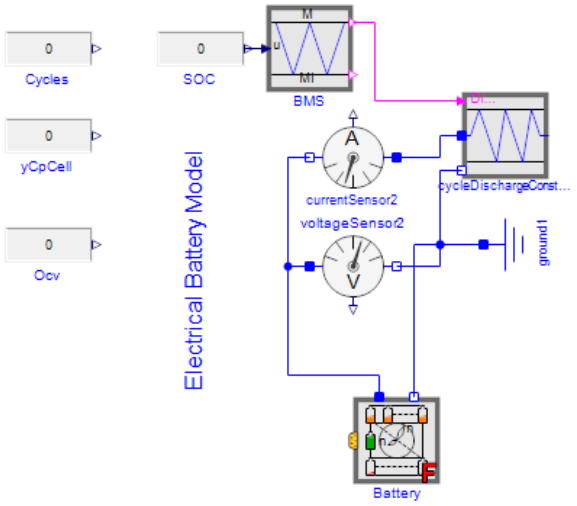

Figure 4. Screenshot of the (generated) faulty model for battery management system tests. The component "Battery" modeled in the battery package contains integrated electrical and thermal faults as well as their combination e.g. because of balancing failure. The battery package parameters are used in HiL for virtual parametrization and management system testing of the battery.

4.3 Diagnosis

The third field of application is concerned with the prediction and identification of (*faults*) in a system. We envisage the case for the non-nominal behavior of a real physical system. Sometimes measurement data from the real system (especially for the non-nominal behavior) is not available or insufficient. This could be because the system is still under development, or because it is a subsystem physically encapsulated by other components and hence inac-

DOI
10.3384/ecp18148161

cessible. In this case the nominal model of the system can be extended to include the non-nominal behavior by including *faults* from the Fault library - but also specific own *fault* models (Ishibashi et al., 2017). This model can be used as a basis for diagnosis on the model itself or only on the (arbitrarily large amount) of data produced by simulations of the model.

As mentioned above, data acquisition in real systems is generally limited, in particular due to a lack of historical data and measurable signals values (Dürr, 2016). In the case where insufficient data from the real system is available, the model may be used to complete the necessary data. In this case, the developed model describes some *faults* behavior of a real system with sufficient accuracy and the subject of validation can be used in place of field tests for generating reference data with various *faults* and time durations. In this way, the produced data allows an appropriate machine learning algorithm to analyze and diagnose *faults* and to identify appropriate *performance indicators* for different *faults* and their combinations.

The model "Feed Axis" in Figure 5 can be used to obtain some of the unavailable signals of the feed drive for the purposes of *fault* diagnosis. Moreover, combining *fault* modeling with the data acquisition of signals in the bus system enables the use of supervised machine learning algorithms for signal *faults* diagnosis. The feature analysis of signals and *faults* with various classification algorithms such as support vector machine or decision tree offers the possibility to choose an appropriate algorithm and to define *performance indicators* for each kind of *fault*.

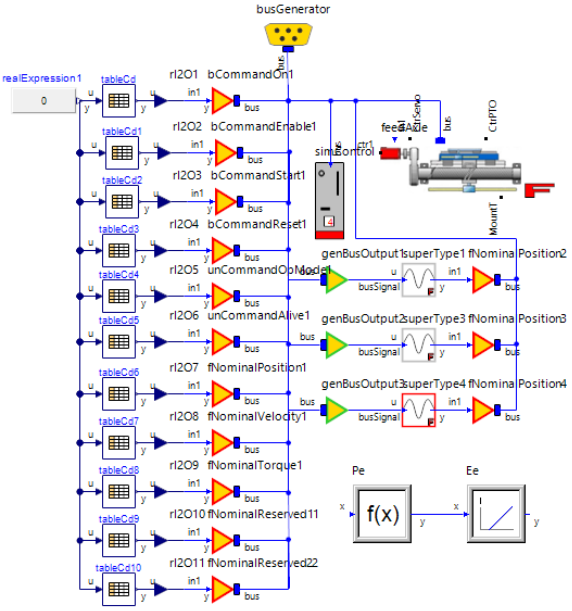

Figure 5. Screenshot of the SimulationX model of a feed axis with faults. The model consists of a feed axis model with servomotor and its control system. The fault signals causes a loss of accuracy and drift in the actual current and position signals in the control in-the-loop.

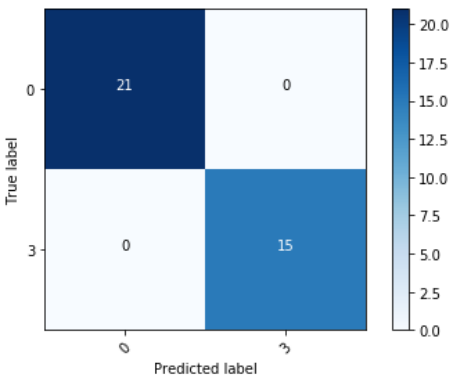

Figure 6. Confusion matrix of faults with decision tree classifier without normalization. The matrix shows identification of 15 faults from 36 test cases and confirms the feasibility of the algorithm configuration.

For example the confusion matrix in Figure 6 shows the identification of drift *faults* in the control system of a feed axis by using the CART algorithm with the Gini impurity metrics. The data is shuffled and split, with 75 percent of it being used for training and 25 percent for testing. The confusion matrix confirms the quality of the chosen parameters in the decision tree for the signal *fault* classification. The decision tree depicted in Figure 7 extracts the importance of features for the signal *faults* identification. As shown in the decision tree, the signal *faults* cause changes in the intensities of the torque and the active power of the feed axis drive which can hence be used as key *performance indicators*.

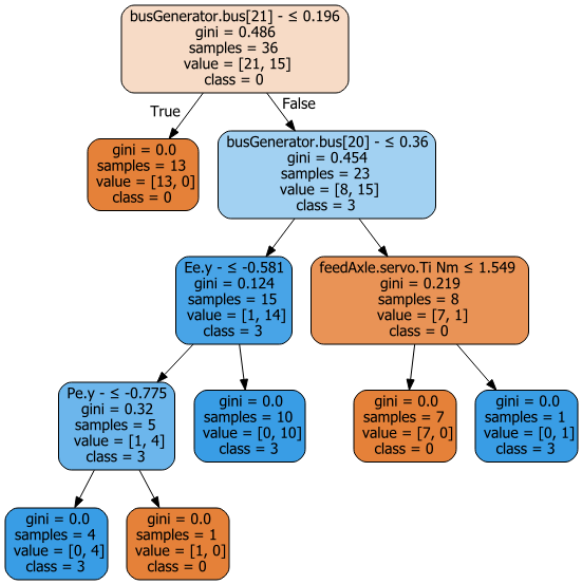

Figure 7. Decision tree of faults diagnosis from bus signals of the feed axis. The decision three shows the key performance indicators for the fault diagnosis in the decision tree - energy consumption, active power, motor torque, etc.

5 Fault Insertion - the FaultAugmenter in SimulationX

Augmenting a model of the nominal system with *faults* can be time-consuming and error-prone. For example, when replacing the nominal type of a component by its fault-augmented counterpart, one has to ensure that the parameters are kept or modified correctly.

To support the augmentation, an AddIn to SimulationX has been developed. A wizard guides the user through the augmentation process. Some input is needed to avoid including too many *faults* - which is helpful for the following analysis. The supported augmentation contains the following steps:

Definition of the candidate: The user has to define the *candidate*, i.e. all those components and connections in the model, which are to be augmented (at most) by drag-and-dropping to a list (Figure 8). Additionally or alternatively he/she can decide whether to exclude or include the augmentation of components (Figure 9 on the left), or the insertion of ConnectorFaults only into connections of specific domains (mechanical, electrical,...).

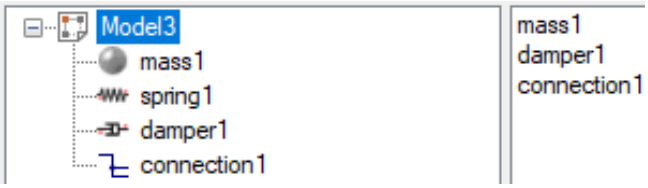

Figure 8. Choosing the components and connections for the fault augmentation with FaultAugmenterAddIn.

Augmentation: The actual augmentation can be started by clicking the button "Augment"- the model structure is changed as shown in Figure 9 on the right. Components have been replaced by their fault-augmented counterparts. To each connected connector a *fault* with two pins ("broken") was added, and one ConnectorFault with one pin ("sticking") was added to the whole connection.

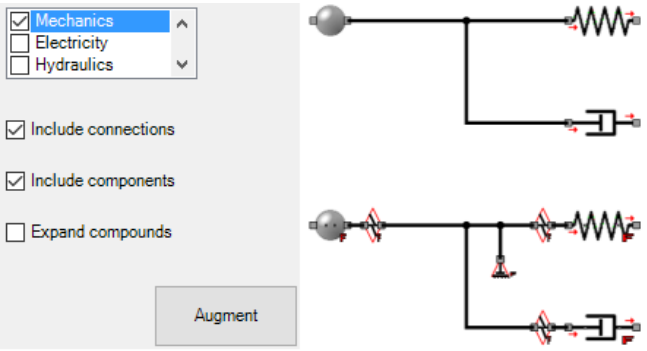

Figure 9. SimulationX model before and after the automatic fault augmentation (right) for mechanical components supported by the FaultAugmenter AddIn (left).

Scaling: The FaultAugmenter AddIn allows to set the `scale` parameter of groups of *faults* or individual *faults* to a certain value (cf. Sec. 3.3). If, for example, the mechanical part of the model represents heavy machinery, the scale in all mechanical connections can be increased by the same order of magnitude. The insertion of BridgeFaults is not conducted by the FaultAugmenter, since on the graph the number of possible additional connections is very high, but BridgeFaults only make sense in very few places. Hence, the user has to insert BridgeFaults manually.

6 Analyzing Faults

As well as structuring the modeling and insertion of *faults*, it is also helpful to structure and systematize the output of the model of interest. In all three fields of application some output quantities of the model are analyzed. However, the goals of the analyses in the different applications are quite different. The Modelica packages presented in this section serve as helpers for these analyses.

6.1 Extracting Features

6.1.1 What for?

In the field of diagnosis the model output is compared to real systems' data. Such data is, in general, sampled, can sometimes be averaged, noisy, or even in frequency domain representation. To prepare the model output in the same way, one needs ways to extract features from the variables.

To avoid dealing with (different) sampling rates and the quality of time series, the diagnosis algorithms based on data presented in Sec. 4.3 - support vector machines, decision trees, etc. - use single-valued data only. To produce such data from output time series, meaningful features have to be extracted.

Furthermore, in the field of reliability system performance is categorized by blocks returning a single value - successful, failed, or undecided/not clear. The determination of these categories (for more details see next subsection) is based on calculations of output variables, which are again features of the latter.

6.1.2 Technical Details

The `Features` package contains helper blocks that support feature extraction. Helper blocks are provided for extracting features such as minimum, maximum, mean, variance, the FFT, or short-term-mean. The list is extended based on the examples studied. One important requirement is that all features are insensitive to numerical side effects. For example, the extraction of the maximum of a variable should not pick a "numerical" peak, the height of which is dependent on tolerance or other numerical artefacts. The extraction of mean values should be possible over restricted time spans to avoid a dependency on the overall simulation time (e.g. the average velocity of the car decreases to zero, because the model driving scenario

DOI 10.3384/ecp18148161

depicts an unnecessary amount of time span after the car has come to rest).

The additional sub-packages `ChecksInFixedWindow`, `ChecksInSlidingWindow`, `SignalAnalysis` from (Otter et al., 2015) - although motivated by a completely different application - serve similar purposes.

6.2 Measuring System Performance

The Braille printer and automotive transmission examples address the question of the system performance when it is subject to *faults*. Stated differently: which *fault* or combination of *faults* leads to the violation of pre-defined criteria. Such a criterion, taken from the transmission example, is: The car shall be able to accelerate from 0 to 100 km/h in 5 Seconds. To assess this in the model, output variables (velocity, time) are read, features are extracted (velocity at 5 seconds after startup) and tested against a criterion (larger than 100 km/h). The formalization of the last step is supported by the elements modeled in the package `PerformanceIndicators`.

The basic definition of a performance indicator contains an array of `assertions` as an input. In the example of the velocity test this array has one entry: $v(t_{Startup}+5)-100[km/h]$. If this entry is greater than zero, the criterion is fulfilled, if not, it is violated. If the array of the assertions contains more than one entry, it has to be defined whether they are connected by an AND (i.e. all must be fulfilled to fulfill the whole criterion) or an OR (i.e. only one must be fulfilled). Sometimes it does not make sense to test a criterion at all - for example, if there was no ignition, there is no startup time. To address this scenario, the performance indicator contains a second input named `validityIndicator` defined in a similar way as the `assertion` - i.e. if this variable is positive, the validity is given and the assertion can be tested, if not, the assertion does not need to be tested. The integer output `perfInd` is based on the validity indicator and the assertions, and is restricted to three values, which represent the categories as listed in Table 1.

For convenience, the assertions are fed to an output variable `assertionsOut`. The output `perfInd` can only tell if the simulation fulfilled or violated the criterion, but not *how far* it was from violating or fulfilling it. To have a measure for this, the relevant continuous variables should be analyzed. Sometimes it is important to have information about how close to violating/fulfilling the specification, e.g. since due to noises/variations which are not in the model the outcome might not be robust. Furthermore, it proves helpful to have some information about whether an increasing *fault* intensity has an effect on an assertion. This information cannot be obtained from the integer output.

Table 1. Categories of the output of the performance indicator based on the incoming validity and insertion.

perfInd	category	conditions
1	fulfilled	validity>0, assertion>0
0	violated	validity>0, assertion<0
-1	undecided	valididy<0

Figure 10 contains components of the `PerformanceIndicators` blocks. Currently, the connection of assertions via AND, OR and NOT is possible, since any logical expression can be brought into either of these forms (disjunctive/conjunctive normal form, see e.g. (Hazewinkel, 1994)). More flexible definitions of performance indicators will be developed as applications demand.

The Modelica_Requirements library presented in (Otter et al., 2015) contains a similar 3-valued logic to those presented here. Its motivation comes from a connection between system simulation and the formal definition of requirements. For the Modelica_Requirements library an extension of the Modelica language is proposed to handle the 3-valued temporal logic (satisfied, violated, undecided). The formalization of proving the logical expressions built up from validity and assertions (as described above) could be improved if the handling of the 3-valued logic becomes part of the language standard. However, it is not necessary in our case.

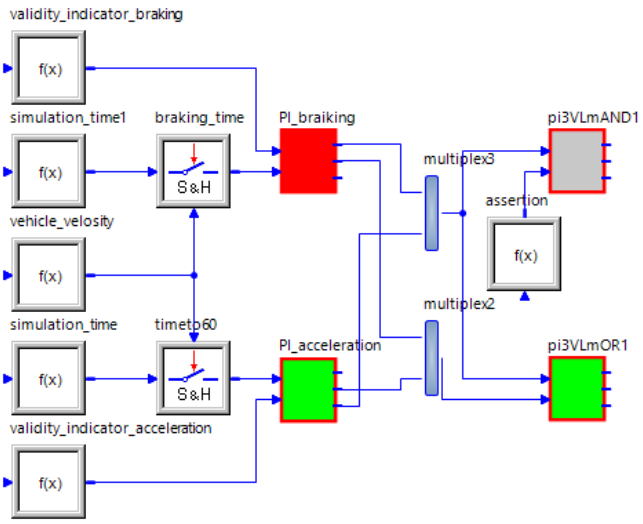

Figure 10. Screenshot of PerformanceIndicators in the SimulationX model of the automatic transmission. The connection of assertion via AND and OR allows to check whether: the car shall be able to accelerate from 0 to 100 km/h in 5 Seconds and to brake from highway speed to stop in 4 Seconds. The fulfillment (green), violation (red) or undecided (grey) states of a performance criterion are displayed.

7 Conclusion

In this publication, we introduced the Fault library , which enables the user to model and simulate physical systems outside their nominal behavior in a systematic way. We motivated the utility of the Fault library with four examples having very different analysis goals. Based on these, the broad range of applicability was outlined. Preliminary

results - based on the given examples - were presented. In addition, the necessity and implementation of helper libraries (Features, PerformanceIndicators) and wizards (FaultAugmenter AddIn) was described.

Acknowledgments

Part of the work lies in the scope of the Romesa project (Robustness and Reliability Simulation of Mechatronic Systems including Aging and Wear (BMBF, 2015)). The authors would like to thank the German Federal Ministry of Education and Research (BMBF) represented by the project coordinator German Aerospace Center (DLR) for supporting financially. Furthermore, thanks are due to Palo Alto Research Center for their development and input on the Fault library, which is based on the FAME library (de Kleer et al., 2013). Thanks go also to Alfred Kamusella, who provided the model of the Braille Printer that served as a basis for the results reported in Section 4 (cf. also Figure 2).

References

Bernd Bertsche, Peter Göhner, Uwe Jensen, Wolfgang Schinköthe, and Hans-Joachim Wunderlich. *Zuverlässigkeit mechatronischer Systeme: Grundlagen und Bewertung in frühen Entwicklungsphasen*. Springer-Verlag, 2009.

Federal Ministry of Education and Research BMBF. *Projektblatt Romesa*, 2015. URL http://www.pt-sw.de/media/content/Projektblatt_ROMESA.pdf.

Johan de Kleer, Bill Janssen, Daniel G. Bobrow, Tolga Kurtoglu, Bhaskar Saha, Nicholas R. Moore, and Saravan Sutharshana. Fault augmented modelica models. In *24th International Workshop on Priciples of Diagnosis*, pages 71–78, Jerusalem,Israel, 2013.

Hans Dürr. In *New Validation Method for Models for Grid Studies*, pages 1–6. Utility Variable Generation Integration Group, 2016. URL https://www.uvig.org/resources/model-validation-workshop/.

Christian Friedrich, Salim Chaker, Christoph Schramm, and Steffen Ihlenfeldt. Generic feed-axis library for machine tools. In *ESI SimulationX User Forum 2016, Dresden, Germany, November 24-25, 2016*, pages 134–143, 2016.

Michiel Hazewinkel. *Encyclopaedia of Mathematics (set)*. Encyclopaedia of Mathematics. Springer Netherlands, 1994. ISBN 9781556080104. URL https://books.google.de/books?id=uxUBQwAACAAJ.

Friedrich Herrmann. Eine analogie zwischen mechanik, elektrizitätslehre, wärmelehre und stofflehre. *PdN PhiS*, 55(2): 2–5, 2006.

Tatsuro Ishibashi, Bing Han, and Tadao Kawai. Rotating machinery library for diagnosis. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 381–387. Linköping University Electronic Press, 2017.

Alfred Kamusella. SimulationX model of a braille printer, 2016. URL https://www.ifte.de/lehre/optimierung/uebung.html.

Raj Minhas, Johan de Kleer, Ion Matei, Bhaskar Saha, Bill Janssen, Daniel G. Bobrow, and Tolga Kurtoglu. Using fault augmented modelica models for diagnostics. In *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 96, pages 437–445. Linköping University Electronic Press; Linköpings universitet, 2014.

Martin Otter, Nguyen Thuy, Daniel Bouskela, Lena Buffoni, Hilding Elmqvist, Peter Fritzson, Alfredo Garro, Audrey Jardin, Hans Olsson, Maxime Payelleville, et al. Formal requirements modeling for simulation-based verification. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 625–635. Linköping University Electronic Press, 2015.

Andrew R. Plummer. Control techniques for structural testing: A review. *Journal of Systems and Control Engineering*, pages 139–169, 2007.

V. Ruiz, A. Pfrang, A. Kriston, N. Omar, P. Van den Bossche, and L. Boon-Brett. A review of international abuse testing standards and regulations for lithium ion batteries in electric and hybrid electric vehicles. *Renewable and Sustainable Energy Reviews*, pages 1427–1452, 2018.

Mugur Tatar and Jakob Mauss. Systematic test and validation of complex embedded systems. *ERTS-2014, Toulouse*, pages 05–07, 2014.

Franciscus L. J. van der Linden. General fault triggering architecture to trigger model faults in modelica using a standardized blockset. In *10th International Modelica conference*, number 96 in Linköping Electronic Conference Proceedings, pages 427–436. LiU Electronic Press, 3 2014. URL http://elib.dlr.de/90576/.

Yinjiao Xing, Eden WM Ma, Kwok L Tsui, and Michael Pecht. Battery management systems in electric and hybrid vehicles. *Energies*, 4(11):1840–1857, 2011.