Publications of the DLR

General fault triggering architecture to trigger model faults in Modelica using a standardized blockset

F.L.J. van der Linden, German Aerospace Center (DLR),
Institute of System Dynamics and Control.
Münchner Straße 20, 82234 Weßling, Germany
Franciscus.vanderlinden@dlr.de

Abstract

The implementation of faults in Modelica is currently not standardized, which leads to many non-compatible implementations. To support the standardization of fault implementations, a new standard for fault implementation and triggering is proposed. The proposed standard can handle parameter faults as well as variable faults during a time simulation to cover all common fault possibilities. Using instance modifiers as well as an inner-outer broadcasting method, the faults can be triggered in a central block. Furthermore, care was taken so that the simulation of the models in a fault-free condition can be guaranteed. A library using the proposed standard was developed. In this library, the fault implementation as well as the triggering of these faults was modeled with the end user in mind. An example implementation is presented which shows the capabilities of the library.

Keywords: Failure, Fault, Modeling, Standardization, Fault Injection

1 Introduction

Failure detection and health monitoring systems to improve reliability and lower maintenance costs become increasingly important. Therefore the design and testing of these algorithms need good prediction models combined with an efficient way to trigger all fault cases.

Implementing faults in Modelica models is no new terrain. Many different implementations of real systems have been made. For example Schallert [8] did a reliability and safety assessment. The faults are automatically identified based on parameter names. To set the parameters of the failed parts, a function is used which automatically sets the parameters before simulation. Gao et al. [3, 2] did a fault analysis of electrical systems. To trigger the faults, two different methods are used; hard coding a fault in the model as well as creating a completely new model for a fault. Cui et al. [1] modeled an actuator system with automatically triggered faults. This automatic triggering is based on the predefined fault probability, but cannot be directly controlled. These works are all examples where faults are triggered in a Modelica implementation. However, all of these implementations use different ways to trigger the faults. Since there is a lack of standards implementation ways, all users must find a way for them self to trigger the faults.

Another approach for model-based diagnosis is used by RODON [7]. Uncertainty intervals for the model parameters combined with behavioral models are used to trigger faults. However, time simulations are not supported which limits its use in many applications. Also FaultWeaver [6] can be used to trigger faults in Modelica. It uses a set of models in Modelica. An external (non-Modelica) program is used to set the faults in Modelica and simulate the results.

In this paper, a set of **standardized** fault-output blocks is proposed in Section 3. These blocks use a designated data type to clearly identify these blocks as special fault blocks for further processing. Using these blocks, it is possible to create component models which include optional faults by the user. Care has been taken to make sure that all possible faults can be modeled by a single or a combination of standardized fault blocks. By analyzing the complete model, built from individual sub-models, a wrapper package can be automatically generated which can be used to activate all faults (Section 5). Care has been taken that it is possible to completely eliminate the fault

code from a model to increase simulation performance if not all faults are triggered. Furthermore, also quick model testing without a fault setup is possible.
The proposed implementation is based on a Dymola implementation making use of the Abstract Syntax Tree (AST) functions from the `ModelManagement` library. By using the proposed **open** and **standardized** fault blocks with a specialized fault type, it is possible to create similar functionalities in all Modelica solvers.

2 Fault injection demands for Modelica

To create a general environment to trigger faults in Modelica, care must be taken that all possible faults can be modeled using the proposed blocks. To make sure all possible faults are covered, a trade off study has been carried out. Fault implementations in the Modelica language can be generalized into classes. The following sections will highlight the different fault classes.

2.1 Fault variability classes

For Modelica usage, two different classes of faults can be identified:

1. Faults that have a very low time constant with respect to the simulation horizon and can be considered constant.
2. Faults with time constants faster than the simulation horizon which will cause transient behavior.

To further clarify the classes, a more detailed description including examples of each fault class is given.

2.1.1 Parameter faults

The parameter fault class consists of faults that have a low time constant compared with the simulation horizon. Usually these faults are characterized by slow changes in time such as the variation of the viscosity of oil due to an aging process in a transmission application. Another example of a fault with a slow time constant compared to the simulation time are some high frequency electronics simulations. In these simulations, the temperature of the environment can often be characterized as constant. Some examples of parameter faults are:

- Gear play
- Degradation of capacitors or batteries
- Oil viscosity degradation in a transmission
- Environment temperature increase in a fast switching application

Due to the very slow nature of these faults with respect to their simulation time, it is not necessary to have the possibility to model fault transients.

2.1.2 Variable faults

The second class of faults are variable faults. These faults are characterized by the possibility that they can significantly change during a typical simulation. Quite often the study of a transient response is one of the main purposes of the simulation of such faults. Some examples are:

- Semiconductor short circuit
- Breakage of hydraulic oil line
- Gearbox tooth breakage
- Screw jam

The faults in this class can vary during a simulation run, and can cause a dynamic system response which might be of interest for the engineer.

2.2 Fault data type classes

The faults of both classes described in Section 2.1, can be divided into three types to represent the different cases needed to model faults.

2.2.1 "On-Off" faults (Boolean)

On off faults are marked by having only two discrete states. Examples are jamming of a nutscrew and disconnection of electrical cables.

2.2.2 Case faults (Integer)

Case faults are marked by having multiple discrete failure modes. An good example is a semiconductor failure:

- Normal operation
- Short circuit
- Open loop

DOI 10.3384/ECP14096427

	Constant	Variable
On-Off	Increased friction	Screw jam
Mode	Bearing fault mode	Transistor
Continuous	Gear play	Oil loss

Table 1: Combined fault possibilities for Modelica models with examples. The choice between a variable and parameter fault is not always directly clear, and may need to be chosen as constant or variable based on the length of the simulation

Flag value	Description
0	fault deactivated
1 (default)	standard fault mode activated
2,3,...	optional extra fault modes

Table 2: Variable mode selection flag

2.2.3 Continuous fault (Real)

A continuous fault is a fault without an explicit discrete value. Examples are:

- Oil degradation
- Increased friction in a bearing
- Capacitor degradation

Combining the fault classes (Section 2.1) and the fault class properties (Section 2.2), six different combinations of faults are identified (see Table 1). These possibilities can model all general and advanced faults. In the next sections, the implementation as well as some extra features for easier fault handling and simulation performance are discussed.

2.3 Variable mode selection

To accommodate the reconfiguration of a model with a variable fault, a mechanism to decide if the fault can be activated during simulation must be implemented. This reconfiguration can be necessary to increase simulation speed in case of no failure or to switch between different failure modes. In the case of a parameter fault, this is known by definition. However in the case of a variable fault, this is not known. To be able to reconfigure such a model, it is therefore necessary to add a parameter signal flag which can be used to reconfigure the model. For maximal flexibility, it is chosen to add a mode selection using an integer constant as a flag. This flag can be used to reconfigure a model to include or exclude a fault. How to use the values of the flag can be seen in Table 2.
The same effect as the mode selection is possible by combining a parameter integer fault with an variable fault. However, combining two fault inputs for one fault makes it hard to use consistent naming.

3 Fault triggering standardisation architecture

Defining faults types is not sufficient to define a usable Modelica implementation. For a good user-friendly implementation a well designed architecture is vital. Different ways to set up a fault triggering method are analyzed and their benefits compared.

3.1 Fault Architecture

Controlling of the faults in a global model using components with faults can be done in many different ways. Different ways are studied in this section and it is decided which methods are selected for the proposed standard.
To assess the overall performance of these methods, a set of criteria is defined to evaluate several important aspects for fault triggering. The implementation effort of setting up the general architecture is not evaluated as this effort has to be invested only once in the generation of the fault library.
These criteria are:

(a) **Non physical connections**: The connections between the models should be based on physical quantities. Faults do not have a physical connections as they are triggered by wear, or external influences which are usually not modeled.

(b) **Ease of implementation**: Effort for user to create a model from instances using faults (e.g. the development of a multistage gearbox using predefined faulty gearbox instances)

(c) **Maintainability**: Effort to maintain a set of models with faults. Typical tasks would be adding or removing fault cases, restructuring models and keeping a well documented set of models

(d) **Standardization**: Standardization effort to keep models compatible between different

business partners.

(e) **Transients**: Possibility to model transients used for variable faults (see for a description Section 2.1.2).

Four methods to implement faults in Modelica have been tried out and analyzed. The results are assessed using the criteria (a through e).

1. **Model parameters**: Each fault is controlled by a parameter in the model. It is possible to "pull" these parameters up to the top level model. Also direct changing of the instance parameter in the model is possible. If the parameter is flagged appropriately, it is possible to create a system with automated parameter detection and central setting of the parameters. Such a structure does not need non-physical connections, is easy to implement for the user and, if a proper automation is used, can be well maintained. Also standardization using the proposed flag methods is possible. Since in this method it is only possible to handle parameters, no transients can be used.

2. **Model inputs**: Inputs are used to control faults in the models. By connecting the input connectors, it is possible to create a central element to control all faults. This method is often used for small fault systems. However, it leads to non physical connections between the models. Due to the high customization, maintainability and standardization cannot be guaranteed. The ease of implementation is good in small systems maintained by a single person, but quickly becomes more and more problematic as the model grows. Transients can be handled well.

3. **Bus system**: A bus system to connect faults. All fault signals are connected to a bus system. This way is similar to the model inputs, except that all faults are organized in one block. A bus system leads to non physical connections. The ease of implementation and maintainability depends highly on the complexibility of the model, small systems can be easily implemented and managed, but it become quickly confusing. The standardization is better than using a direct model input since all faults are now marked in one fault-bus. However, still no automated algorithms can be used as it is impossible to properly define a standard input. Transients can be handled well as it is possible to connect variables to a bus.

4. **Broadcasting**: Using an inner-outer structure, the fault models can obtain their values from a centralized point in the global model. Using an automated routine, all appropriate flagged faults can be found and managed in one central point. No non-physical connections are needed. If standardized, flagged and predefined fault blocks are used, the ease of implementation and maintainability is high. Also the standardization can be guaranteed by flagging the models. Using an inner-outer structure it is also possible to use transients. However, when only parameter Faults are used, this way of modeling is over complex and will always need a full setup of the variable faults. In contrast, it is possible to leave most parameter faults at their standard value and set only one fault without setting up a complete fault system.

In Table 3, the previously discussed four different approaches (1:4) are assessed using the criteria (a:e). From this analysis follows that the usage of model parameters (1) and a broadcasting system (4) have most advantages. It is therefore chosen to use following architecture:

- **Model parameters** to handle **constant** faults
- **Broadcasting system** for **variable** faults

4 Standardized fault class definition

All faults in a model must be recognized by automated scripts, while at the same time the user should have the freedom to name the model faults arbitrarily. To do so, special fault classes have been designed. This has the advantage that a fault is identified by the class name, and can be integrated without special care of instance names by the user. These fault classes are released under the Modelica 2 License.
Below the all fault classes are defined. For a parameter fault of the type Real, the type is defined in Code 1.

DOI 10.3384/ECP14096427

Description	(a) Non physical connections	(b) Ease of implementation	(c) Maintainability	(d) Standardization	(e) Transients
(1) Model parameters	+	+	+	+	-
(2) Model inputs	-	±	-	-	+
(3) Bus system	-	±	±	±	+
(4) Broadcasting system	+	±	+	+	+

Table 3: Fault triggering approaches. A detailed description of the criteria can be found in Section 3.1

.

Code 1: Real parameter fault

```
type Parameter_Fault_Real =
        Real "Value of the Real Fault";
```

Using this special fault class for each Real fault, it is possible to clearly identify each instance of this model as a Real parameter fault.
The same is done for variable faults. Since these faults are more complex, a record with three parameters is used. In Code 2 the definition of this Fault is shown.

Code 2: Real variable fault

```
record Variable_Fault_Real
   "External Fault Triggering parameters"
  Boolean externalFaultOn=false
   "External fault controlling
                 (true = global)";
  Integer faultIndex = 1
   "External fault index";
  Integer faultMode = 1
   "Optional fault mode for model
                 reconfiguration";
end Variable_Fault_Real;
```

The first Boolean `externalFaultOn` is used to switch between the local default fault definitions and external global control. The integer `faultIndex` is used to set the channel in the external global fault triggering (see Section 5). The Integer `faultMode` is used to set the optional fault mode selection as discussed in Section 2.3.

The examples given in this section are for Real faults. The code for Integer and Boolean faults can be found in Appendix A.
Using these class definitions, it is possible to set up a complete fault triggering system. In Section 5 an implementation for Dymola using the ModelManagement toolbox is presented. Since these defined faultclasses are open and standardized, algorithms or plug-ins for programs can be developed by users, also for other Modelica solvers.

5 FaultTriggering library

Beside the definition of a standard, a library has been built to support the user with implementing faults. Using the definitions from Section 4, blocks are created which simplify the implementation of faults in a model. Two versions of these outputs are made; one for textual modeling and one for usage in the diagram layer. Also a method to manage the fault signals in a single block is proposed.
In Figure 1, an overview of the final fault setting structure is given. In the generated wrapper model, it is possible to set the parameter and variable faults. The parameters are set in an automatically generated structure (see Section 5.3.2) and the variable faults are handled using a generated bus system (see Section 5.3.3).

5.1 Parameter fault modeling

The textual modeling block for a parameter fault is a simple block with a parameter of type `Parameter_Fault_Real` (for Real faults). By extending this block in the model, a parameter fault is directly correctly implemented and its name is `constRealFault`.
In Code 3 the code for the Real parameter fault is given. Parts of the complete path to the components are abbreviated for a better overview. Integer and Boolean faults are implemented using the same approach.

Code 3: Real parameter fault for textual modeling

```
block InternalConstantRealFault
 "Generate constant Fault of type Real"
 extends ...Icons.RealFault;
 parameter ...Types.Parameter_Fault_Real
      constRealFault= 1
      "Constant output value";
end InternalConstantRealFault;
```

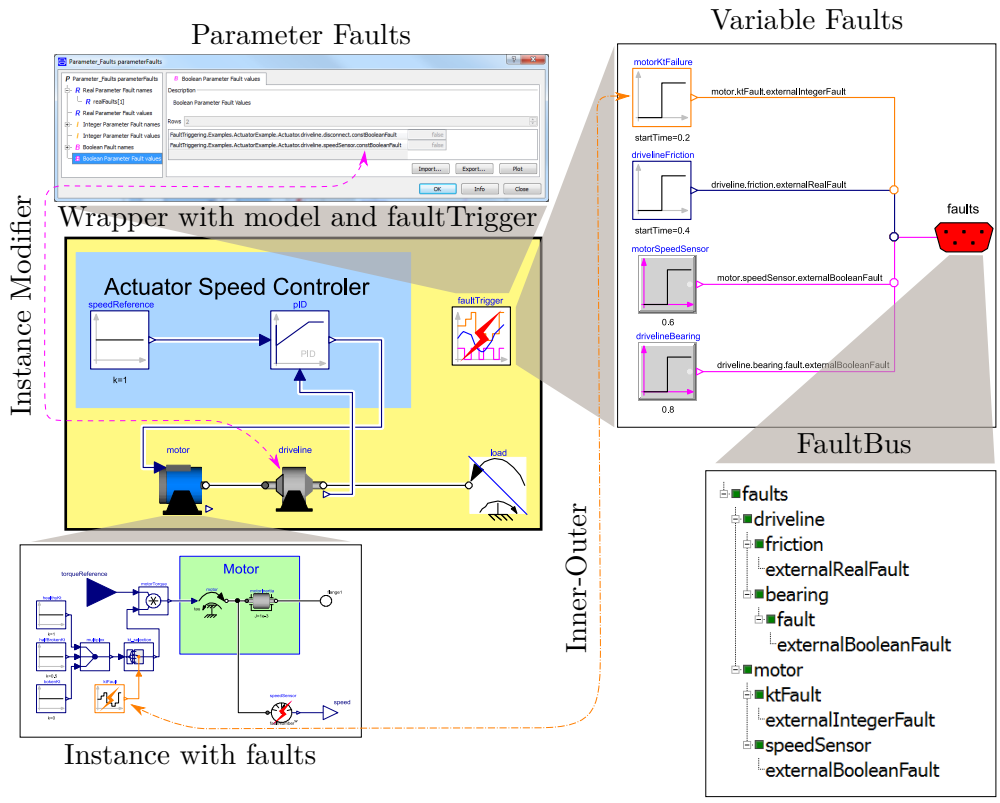

Figure 1: Automatically generated wrapper model (yellow) which contains the extended original model and the block `faultTrigger`. In this block all parameter and variable faults can be set. The parameter faults communicate directly with the model instances using instance modifiers (pink dash-dotted line), the variable faults using a bus system connected to a global inner/ outer system (orange dashed line).

The blocks for graphical modeling environment are extensions of the discussed textual modeling approach and the interface `Modelica.Blocks.Interfaces.SO` . This creates a block (see Code 4) with a single Real output whose value is set by the fault parameter.

Code 4: Real parameter fault for graphical modeling

```
block ConstantRealFault
  "Generate constant signal of type Real"
  extends Modelica.Blocks.Interfaces.SO;
  extends ...InternalConstantRealFault;
equation
  y = constRealFault;
end ConstantRealFault;
```

5.2 Variable faults

The variable fault blocks are more complicated to implement than the parameter fault blocks. These blocks need the information from a central block in which the fault signal is defined. To do so an inner-outer structure has been set up to communicate the fault signals. In this section, first the global block will be discussed followed by the local fault blocks.

5.2.1 Global variable faults control

For each variable fault (Real, Integer and Boolean), a single channel is reserved in a variable with n channels (with n the number of faults of each type). This variable is defined in a central `FaultTrigger` block `extended from ...FaultOutput.Partial_FaultTrigger`. Each fault can be coupled to fault sources using modelica code in this global block. This can be done by hand or an automated script which is proposed in Section 5.3.3. The partial model can be seen in Code 5. This model is defined as "inner" in the annotations, so that the local fault injection blocks can communicate with this block.

Code 5: Partial model for variable fault input framework

```
partial model Partial_FaultTrigger
  "partial model defining  fault classes"
  parameter Integer realFaultSize
    "Number of real fault channels";
```


DOI 10.3384/ECP14096427

```
  parameter Integer integerFaultSize
    "Number of integer fault channels";
  parameter Integer booleanFaultSize
    "Number of boolean fault channels";

  Real    realFault[realFaultSize]
            "Real Fault trigger";
  Integer integerFault[integerFaultSize]
            "Integer Fault trigger";
  Boolean booleanFault[booleanFaultSize]
            "Boolean Fault trigger";

 annotation (
 defaultComponentPrefixes="inner")
end Partial_FaultTrigger;
```

5.2.2 Variable fault modeling classes

The variable fault models get their signals from the global fault control model. In Code 6 the code for a variable fault is given. In this model, the variable `fault` is the actual fault value. Each fault uses its own fault channel in the variables `realFault`, `integerFault` and `booleanFault`. To select which channel is to be used from these variables, the parameter `faultNumber` is defined. This parameter is generally set by an automated system (see Section 5.3.3).
To be able to directly operate a model with variable faults in the model for testing purposes, a parameter with a default fault value is defined in `fault_local`. Using the switch (`externalRealFault.externalFaultOn`), the local fault can be changed to the value defined in the global fault block. Using this, it is guaranteed that each block has a valid output without setting up a global fault block.

Code 6: Real variable fault for textual modeling

```
block InternalRealFault
 "Generate variable Fault of type Real"
 extends ...Icons.RealFault;
 outer FaultTrigger faultTrigger;
 parameter Real fault_local = 1
  "Default fault value if no external
   triggering is used";
 parameter ...Types.Variable_Fault_Real
     externalRealFault =
       ...Types.Variable_Fault_Real()
  "External Fault Triggering parameters";
 Modelica.Blocks.Interfaces.RealOutput
   fault "Final fault value";

protected
...Types.Fault_SelectRealFault
        faultNumber;
equation
 faultNumber =
   externalRealFault.faultIndex;
 fault =
  if externalRealFault.externalFaultOn
  then
  faultTrigger.realFault[faultNumber]
  else fault_local;
end InternalRealFault;
```

The faults for graphical modeling are made by extending Code 6 in a model with two outputs (Code 7): One real output for the fault signal and one optional integer output for the mode signal.

Code 7: Real variable fault for graphical modeling

```
block VariableRealFault
   "Generate variable signal of type Real"
extends ...Internal.InternalRealFault;
parameter Boolean useModelModeSelection
   "toggles external mode selection";
Modelica.Blocks.Interfaces.RealOutput y;
Modelica.Blocks.Interfaces.IntegerOutput
    mode = externalRealFault.faultMode
       if useModelModeSelection
    "Connector of Integer output signal";
equation
  y = fault;
end VariableRealFault;
```

5.3 Automated fault handling

To keep overview of the faults in a model and help the user with fault channel selection for each fault, an automated fault handling algorithm is developed. This algorithm can detect the parameter and variable faults in the selected model. Also all faults in the instances used in this model can be detected. Setting and internal handling of these faults is different for parameter and variable faults. A library is automatically generated which contains a wrapper model that extends the original model. Also a central block to manage the faults is instantiated in this wrapper model. In this block, the configuration of the parameter and variable faults is handled.

5.3.1 Automated fault finding

Using the Dymola `ModelManagement` toolbox, it is possible to investigate a model with its submodels. Using these features, it is possible to generate a model tree from a model with all instances. A schematical example of such a model tree can be found in Figure 2. Using the type definitions from Section 4, all faults in a model can be found and

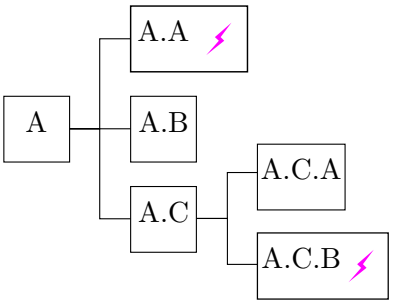

Figure 2: Model tree with faulty models on several levels. Blocks with a lightning symbol (A.A & A.C.B) are extensions of the standardized fault classes.

classified. Also the "path" to the model can be found. The complete path of the fault instance `bearing_stuck` in the model `Actuator` will be represented as `Actuator.bearing_stuck`. In the following sections, the generated fault tree and fault classification will be used in the implementation of the library features.

5.3.2 Global parameter setting

All the parameter faults can be found and identified using a method described in Section 5.3.1. It is possible to directly change these values by using an instance modifier generated by the fault-search algorithm by hand.
However, in case of large models with many faults or many different cases to analyze, this can quickly become unclear and tedious. To aid the user, a structure is automatically generated using the scripts supplied in the library which includes all faults together with their default values. This structure is used as a parameter in the global `faultTriggering` model. These values are automatically linked to the instance modifiers in the wrapper model. By creating different fault structures, fault cases can be defined. Each fault structure stands for a clearly defined simulation case.

5.3.3 Global variable setting

To aid the user with setting the variable faults, a hierarchical `faultbus` is generated from the fault structure (see Figure 1). It is possible to directly connect the fault source signals to the hierarchical bus. The hierarchical bus system itself is connected to the `realFault`, `integerFault` and `booleanFault` variables (see Section 5.2.2). The corresponding fault index is automatically set in

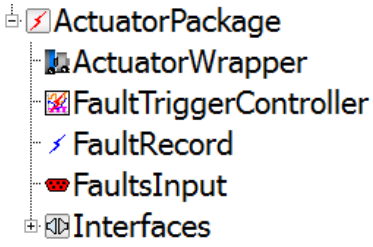

Figure 3: Automatically generated Fault library

the model using component modifiers. Using this approach, mistakes with mixing up the channels are not possible, as this is automated. Also the use of an automatically generated bus makes connecting the fault sources easy.
In Figure 3, a generated library is shown with its default components.

6 Examples

To test the library functionality a simple actuator model is built consisting of a motor with PID control, a simple driveline and a load. The total model has 6 faults: 2 parameter faults, and 4 variable faults. Using the fault processing algorithms presented in Section 5.3, a package is generated. The model wrapper adds the `faultTrigger` block in which all faults can be set. In this block all fault inputs are defined. The variable faults are set in the block of type `FaultTriggerController` (instance `faultTrigger`). Using the parameter record in this block, it is possible to set all parameter faults. An overview of this functionality is shown in Figure 1.
The result of a simulation with progressive faults is shown in Figure 4. The dynamic effects of a breaking component can be seen by the changing response of motor speed and torque. By changing or duplicating the faultTrigger block, it is possible to create multiple fault scenarios for a single model. The original model stays unchanged and can be used for all analysis, healthy as well as broken.

7 Conclusion and Discussion

In this paper, a method to standardize the implementation of faulty components in Modelica is specified. It is possible to implement parameter, as well as varying fault signals. The code for the proposed faulty components is included to aid the

DOI 10.3384/ECP14096427

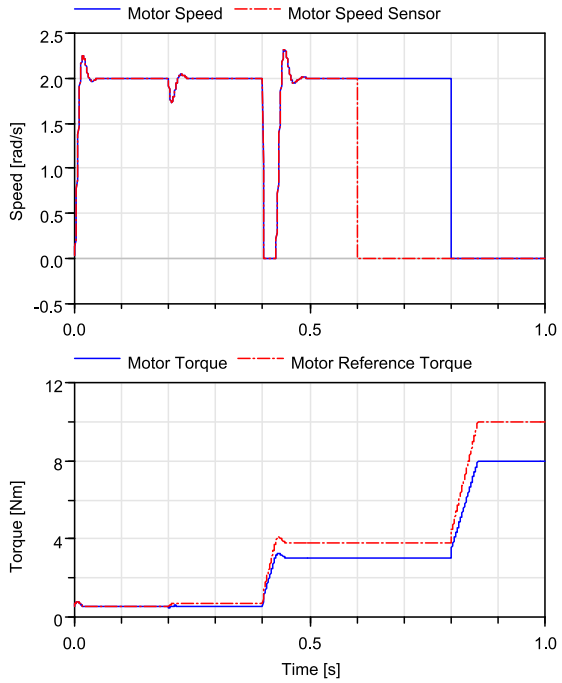

Figure 4: Results of a simulation with progressive faults. At t=0.2s the motor constant drops, at t=0.4s, an increased friction in the driveline is triggered, at t=0.6s, the speed sensor of the motor breaks and finally at t=0.8s the driveline bearing gets stuck.

standardisation of fault implementation in Modelica.

Moreover a library has been created which supports the user to set these faults by automatic generation of a wrapper library. This wrapper includes all parameter faults in a parameter structure and a bus system to connect the variable faults. This functionality is enabled by the implementation of a search algorithm to search the model for the standardized fault classes.

An example model has been built and the methods to implement the faults in the model have been proved valuable. At the moment the proposed Fault Library is used in the Actuator EMA library [4, 5, 9]. The standardization of these faults has led to an easy implementation process. The model designer can focus on implementing the faults in the model without paying attention to the interfaces and the compatibility between the models.

Acknowledgement

The research leading to these results has received funding from the European Union's Seventh Framework Program (FP7-284916) for ACTUATION 2015 under grant agreement no. 284915.

A Modelica Code for Faults

The code for the implementation of the fault classes is given in this section. Using strictly this code, it is possible for automated fault systems to search for all faults in a model.

A.1 Real faults

Code 8: Real parameter fault

```
type Parameter_Fault_Real =
        Real "Value of the Real Fault";
```

Code 9: Real variable fault

```
record Variable_Fault_Real
   "External Fault Triggering parameters"
  Boolean externalFaultOn=false
   "External fault controlling
                  (true = global)";
  Integer faultIndex = 1
   "External fault index";
  Integer faultMode = 1
   "Optional fault mode for model
                  reconfiguration";
end Variable_Fault_Real;
```

A.2 Integer faults

Code 10: Integer parameter fault

```
type Parameter_Fault_Integer =
    Integer "Value of the Integer Fault";
```

Code 11: Integer variable fault

```
record Variable_Fault_Integer
   "External Fault Triggering parameters"
  Boolean externalFaultOn=false
   "External fault controlling
                  (true = global)";
  Integer faultIndex = 1
   "External fault index";
  Integer faultMode = 1
   "Optional fault mode for model
                  reconfiguration";
end Variable_Fault_Integer;
```

A.3 Boolean faults

Code 12: Boolean parameter fault

```
type Parameter_Fault_Boolean =
    Boolean "Value of the Boolean Fault";
```

Code 13: Boolean variable fault

```
record Variable_Fault_Boolean
   "External Fault Triggering parameters"
  Boolean externalFaultOn=false
   "External fault controlling
                 (true = global)";
  Integer faultIndex = 1
   "External fault index";
  Integer faultMode = 1
   "Optional fault mode for model
                 reconfiguration";
end Variable_Fault_Boolean;
```

References

[1] Cui, X., Ma, J., and Zeng, S. The fault modeling methodology of actuator system based on Modelica. *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety* (June 2011), 997–1002.

[2] Gao, J., Ji, Y., Bals, J., and Kennel, R. Fault Detection of Power Electronic Circuit using Wavelet Analysis in Modelica. In *Proceedings of the 9th International MODELICA Conference* (Munich, Germany, Sept. 2012), no. 76, pp. 513–522.

[3] Gao, M., Hu, N., Qin, G., and Xia, L. Modeling and fault simulation of propellant filling system based on Modelica/Dymola. *2008 2nd International Symposium on Systems and Control in Aerospace and Astronautics* (Dec. 2008), 1–5.

[4] Giangrande, P., Hill, C., Gerada, C., and Bozhko, S. Multi-Level Library of Electrical Machines for Aerospace Applications. In *Proceedings of the 10th International Modelica Conference* (2014).

[5] Hill, C., Giangrande, P., Gerada, C., and Bozhko, S. Implementation of a Multi-Level Power Electronic Inverter Library in Modelica. In *Proceedings of the 10th International Modelica Conference* (2014).

[6] Junghanns, A., Mauss, J., and Tatar, M. TestWeaver - A Tool for Simulation-based Test of Mechatronic Designs. In *Proceedings of the 6th International Modelica Conference* (2008), pp. 341–348.

[7] Lunde, K. Object-oriented modeling in model-based diagnosis. *Proceedings of Modelica Workshop, Lund, Sweden* (2000), 111–118.

[8] Schallert, C. Inclusion of Reliability and Safety Analysis Methods in Modelica. In *Inclusion of Reliability and Safety Analysis Methods in Modelica* (June 2011), pp. 616–627.

[9] van der Linden, F., Schlegel, C., Christmann, M., Regula, G., Hill, C., Giangrande, P., Maré, J.-C., and Egaña, I. Implementation of a Modelica Library for Simulation of Electromechanical Actuators for Aircraft and Helicopters. In *Proceedings of the 10th International Modelica Conference* (2014).