J. Köhler, A. Banerjee
ZF Friedrichshafen AG, Germany

Usage of Modelica for transmission simulation in ZF
pp. 587-592

Usage of *Modelica* for transmission simulation in ZF

Jochen Köhler Alexander Banerjee
ZF Friedrichshafen AG
Graf-von-Soden-Platz 1, D-88046 Friedrichshafen, Germany

Abstract

At ZF transmission models are an essential component within the development process. To guarantee an efficient use of modeling know how as well as a persistent use of models in different simulation environments, Dymola has been declared as a standard at ZF and a central model component library ZFlib has been developed accessible to all business units. This paper gives an overview of some features of the library, such as easy parameterization of models independent of the environment in use and the export of complex models into environments with simple integration algorithms. Furthermore a short description of the automatic testing of the library which guarantees a software development process at a high level of quality will be given.

Keywords: Model Export, Parameterization, Automatic Testing

1 Introduction

For ZF as an automotive vendor for transmission components and systems, modeling and simulation has a long tradition. A long time ago the modeling of transmission systems was mainly done by experts of the business units which often used different tools and different approaches. In the meanwhile there has been an exponentially increasing demand for models and model components which should be accessible to and usable even by non-experts. Due to this fact ZF started a centralization process with the intention to standardize the modeling approach of transmission systems and merge the modeling know-how of components and systems in one library accessible to all business units. A further requirement has been to guarantee a persistent use of models in different simulation environment (e.g. *SIMULINK* or *dSpace*), which actually demands for models given as C-Code (Figure 1). Beside this, the tool under consideration

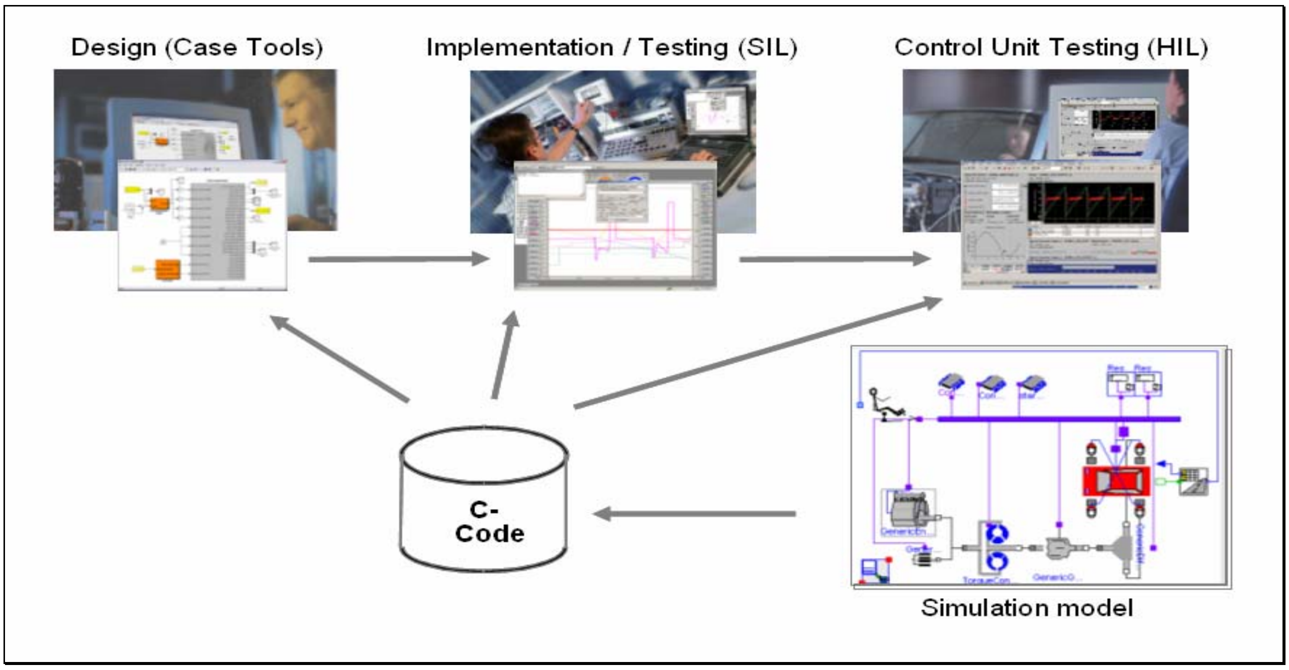


Figure 1: Exporting models to various environments

should offer the user an interactive graphic interface for modeling. Since Dymola offers such an interface, allows the continuous extension of libraries by using the Modelica language and enables the user to export models as C-Code, ZF came to the decision to define Dymola as a standard for transmission system modeling.

This was the starting point for the development of a *Modelica* library called `ZFlib` which is an extension of the commercial library `Powertrain`. This library has a lot of models of ZF specific components as well as some add-ons for the export to other simulation tools.

The focus of this article is a short explanation of the ZFlib and how models can be parameterized independently from the environment in use (chapter 2), how the export of complex models into environments which only allow the use of discrete integration algorithms (chapter 3) has been realized and how an automatic testing of modules can be performed whenever a modification within the library has occurred (chapter 4).

2 Usage of `ZFlib`

2.1 `ZFlib` as code generator

The main issue in using the `ZFlib` is to set up models that can be used within different environments – not only with *Modelica* Tools like *Dymola* or Math-*Modelica*. *Dymola* is used to translate the models developed with *Modelica* to C code.

The generated C code in form of the `dsblock` function can be integrated into the demanded environment. This is done already by Dynasim for *Simulink* with a special interface-module called DymolaBlock.

Nevertheless we did some modifications of this module to meet our special demands. Other interfaces for further environments (e.g. ASCET and some ZF programs) were developed to wrap the different calls (e.g. initialization, update …) from these environments to the appropriate functionality of the `dsblock` function.

One big drawback with the generated code of a standard *Modelica* model is that we can't parameterize the model in *one*, easy way in different environments. Another aspect is, that we want to use a standardized way of parameterization in ZF for these models. The basic idea was to separate models and parameters from each other. The parameter values should be stored in ASCII-files according to a standardized format. At initialization the files are read by the model for parameterization. Therefore some extensions had to be done to read the files and parameterize the modules in an easy way.

2.2 Performing parameterization in different environments

The format of these ASCII-files has a very primitive syntax to define scalar or vectors as well as characteristics with variable dimensionality. One parameter is defined simply by a triple: Identifier, Unit and Value (see Figure 2).

```
J1   [kgm^2]  0.1
; scalar parameter
InU  [-]      0   1   2
OutY [-]      0   1   2
; two vectorial parameters
Test_Table2D[
[-]   U1 [-]  0   1    2
2     Y  [-] -2  -1    0
1     Y  [-] -1   0    1
0     Y  [-]  0   1    2
Test_Table2D]
; Two-Dimensional-Table
```

Figure 2 Example of an ASCII-file

A large database of parameter data exists already in ZF that shall be used also for the *Modelica* models. We implemented a *Modelica* package to read these files and made the data available to other modules.

The functionality consists of two parts:

First, we implemented some basic *Modelica* functions that serve as an interface to a set of C-functions, which do the data-management.

Second, the C-functions handle the reading of the needed files, collect all parameter data specified there and make them available within *Modelica*. For tables, also the interpolation is done in C-Code.

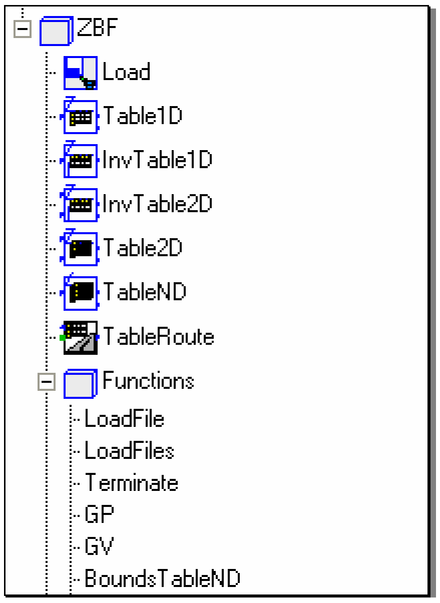
To use this functionality, users have to do the following things:

1. Specify the files to be read:

By including the `Load` block, the user specifies all the files to be read in a list. At initialization all parameters are scanned and stored in database.

2. Use scalar or vectorial parameters

With the ZFlib-functions `GP(String identifier)` and `GV(String Identifier)`, the model finds the data of interest through the defined data identifier within the database. If the asked identifier is not found in the database, the simulation aborts.



ZBF
- Load
- Table1D
- InvTable1D
- InvTable2D
- Table2D
- TableND
- TableRoute
- Functions
  - LoadFile
  - LoadFiles
  - Terminate
  - GP
  - GV
  - BoundsTableND

Figure 3 ASCII file package in the `ZFlib`

3. Use tables

For using tables we have implemented three different Table blocks for one-, two and N-dimensional Tables. They also get access to the tables within the database through some interface functions. With the function (`BoundsTableND(String Id)`) the boundaries of the table input vectors can be queried. In addition to the interpolation, the gradient (one dimensional) or normal vector (n-dimensional) can be calculated on demand. This is very useful for some forward control algorithms.

Another useful feature is the possibility to convert the ASCII data to SI units. This is needed because very often data are saved in units like [rpm] or [mm] and *Modelica* handles only SI units.

The implementation of these features was done in C++. The sources or binaries can be included in various environments and therefore, this kind of parameterization can be used everywhere, where code can be included.

Using this functionality allows the user of the model to change the parameters easily by editing the used ASCII files. The changed parameters will be used at the next simulation run without a new translation of the model.

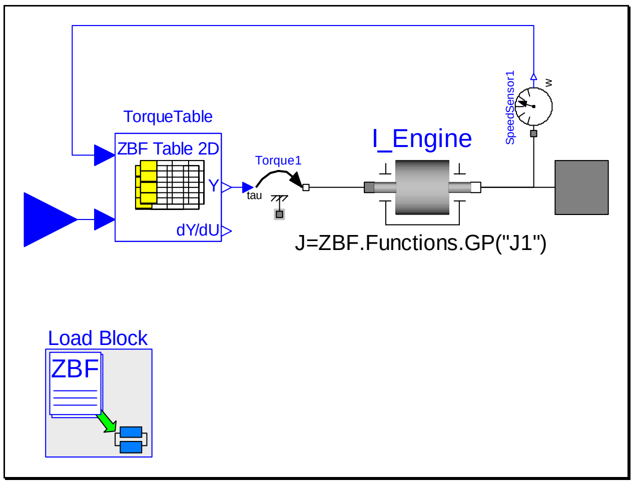


Figure 4 Example of a model using ASCII file package

2.3 Extending from the `Powertrain` library

The `ZFlib` was extended from the commercial library `Powertrain.` The maximum benefit can be achieved by reusing the included elements as often as possibly. Nevertheless, we want to use our own tables within the models in the `Powertrain` library. Due to this, modifications have to be done for a few models within the `Powertrain`: The *Modelica* keyword `replaceable` in front of all declarations of tables had to insert.

Another modification concerns the usage of the bus. The concept of the `Powertrain` bus has a lot of advantages: The implementation of complex models is easily done and quite transparent. It's very easy to change big parts of a complex model without intensive modifications. We decided to extend our own bus called `ZFBus` from the Bus in the `Powertrain`. To be able to use the control units in the `Powertrain` together with this extended bus it was also necessary to add the `replaceable` keyword to all bus declarations with control unit implementations. It would be more convenient, if it would be possible in *Modelica* to link connectors that extend from one root. Another approach to make combinational usage of different libraries easier could be a hierarchical structure of one global bus.

With this second modification we can use all models in the `Powertrain` and our library seamlessly. Certainly it would be a great help to add these modifications to the original `Powertrain` without any disadvantage for other users.

3 Using *Modelica* models with discrete algorithms in *Simulink*

Simulink is used more and more to develop control algorithms for transmissions in ZF. The simulation of systems with a digital controller requires the use of a discrete, fixed step integrator. On the other hand, there is a very strong demand to use complex and often numerically stiff models of transmissions to test these controls in simulation. Hence the use of numerical integrators with varying step size is necessary.

These transmission models are modeled in *Dymola* using the `ZFlib`. Because of the arguments mentioned above, the original DymolaBlock for *Simulink* can't be used unmodified in conjunction with the fixed step integrator from *Simulink*.

3.1 Making the Dymolablock "discrete"

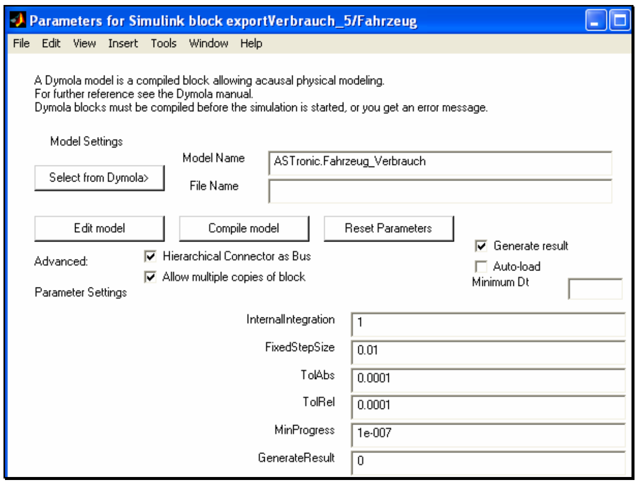
So, we decided to embed a continuous variable step integrator within the S-function of the DymolaBlock. We used the same DASKRT integrator as in *Dymola* [1]. To make the modified DymolaBlock look like a discrete Block, the S-function just returns that no continuous states are needed. Then it's easily possible to combine this model with discrete controllers in Simulink.

The DASKRT is used exclusively for the exported Modelica model inside the S-function. Every time the outputs are demanded from Simulink, the S-function calls the internal integrator to integrate up to the new time. During this interval several events may happen, that have to be handled. Additionally we have to make sure, that the DASKRT stops correctly at the given time point. Otherwise it may happens, that the integrator takes a step size larger than the sample interval of the *Simulink* model. This would lead to outputs which refer to a later point in time with respect to the actual integration time.

3.2 Controlling the behavior of the DymolaBlock

A simple model was implemented in *Modelica* as a parent to be able to control the behavior of the modified DymolaBlock. The declared parameters can be read by the DymolaBlock. The user can enable or disable the internal integration by setting the parameter InternalIntegration to 1 or 0. So it's possible to use the model also in continuous environments or with the simple fixed step integrators of Simulink.

The FixedStepSize specifies the sample time of the block. Furthermore you can specify the tolerances of the integration algorithm. MinProgress is an abstract criterion to abort the simulation if the simulation progress is too low.


Parameters for Simulink block exportVerbrauch_5/Fahrzeug

A Dymola model is a compiled block allowing acausal physical modeling.
For further reference see the Dymola manual.
Dymola blocks must be compiled before the simulation is started, or you get an error message.

Model Settings

Model Name: ASTronic.Fahrzeug_Verbrauch

File Name:

Advanced: [✓] Hierarchical Connector as Bus; [✓] Allow multiple copies of block

[✓] Generate result; [ ] Auto-load; Minimum Dt:

Parameter Settings

InternalIntegration	1
FixedStepSize	0.01
TolAbs	0.0001
TolRel	0.0001
MinProgress	1e-007
GenerateResult	0

Figure 5 The user can change the DymolaBlock behaviour easily by changing the mentioned parameters

With GenerateResult the user can enable the logging to the dsres.mat file if needed. This is more convenient than the check box in the original DymolaBlock because you can change it without compiling.

The interface between *Simulink* and the *Modelica* model consists of two connectors that contain all input or output signals. It's possible to use the same or similar connectors for different models because they contain the same signals as the "real software connectors" in a vehicle. Another advantage is that existing algorithms can be connected easily to these models.

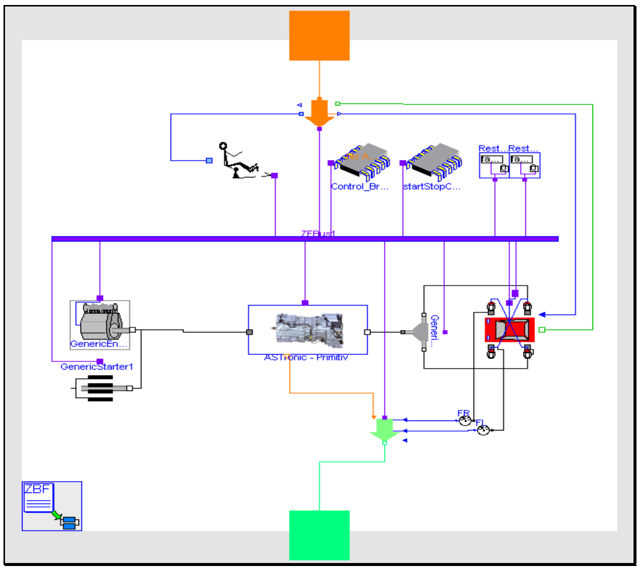


Figure 6 Example for a model that can be exported to *Simulink* with one input connector (top) and one output connector (bottom).

As one can see in Figure 6, the interface between *Simulink* and *Modelica* consists exclusively of directed input or output signals. This is caused by the analogy to a connector for a control unit but it's also a design decision to have no coupling on a "physical layer". Due to this fact, there is no problematic coupling of two integrators as it can be with "normal co-simulation".

4 Automated Testing of `ZFlib` components

One important aspect in implementing basic elements for a library which will be used heavily in simulation, is to be sure, that everything works correctly. Based on experience, modifications on library elements that worked well before can cause strange errors in a complex model. To avoid this, we introduced basic "single element tests". For (almost) every ZFlib element at least one test exists. Such a test shall stimulate an element in a way that a specific behavior can be checked.

For more complex elements, it is necessary to set up several tests with different parameters or with different structures to cover all possible "areas". After the tests have been set up, the developer has to judge, if the simulation result is correct. This result and the corresponding parameters are saved as a reference for later comparison. Of course, this work has to be done very careful!

This approach leads to several benefits. The original developer had to make sure that the new module works as required. The second benefit is the possible use of the tests as an example on how to use the tested element. As mentioned, the third benefit is the possibility to test this element after modifications against the old references.

Also, if a developer finds a bug in an element later on, he can set up new tests that check, if the bug still exists. Testing with these new tests, the developer can be sure, that the same error can't pop up again after another modification.

These tests are useful only, if they are run on regular basis. They should be performed and automatically compared with the reference results of all elements. Therefore we implemented some Matlab scripts, which scan the *Modelica* code for existing tests, run them automatically and compare the results with the reference files.

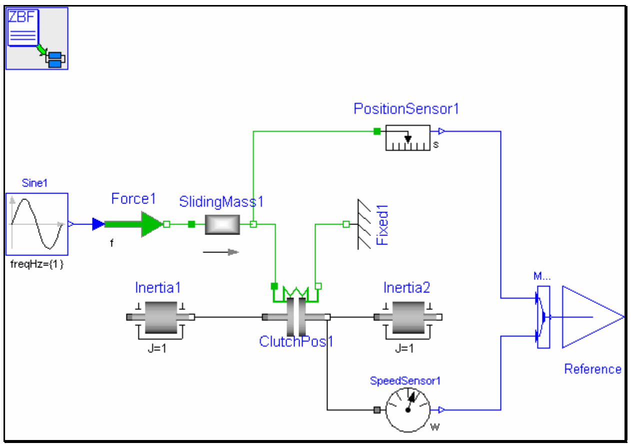


Figure 7 Test for a clutch with an opening/closing spring.

4.1 Collecting tests

To add a test to the automatic procedure it must fulfill two conditions: It has to be inside a package called Tests and it has to be extended from the basic model `ZFlib.Tools.TestModel`. This model defines some tolerance parameters and an `OutPort` reference to the signals which are used for comparison.

```
// preparation
clear
openModel("../dymola/zflib/package.mo")
checkModel("ZFlib")
cd ../dymola
// newModel M_Limiter
translateModel(
"ZFlib.Blocks.NonLinear.Tests.M_Limiter")
// newResultTest
RunScript(
"../TestReferences/M_Limiter_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_Limiter.mat
// endTest
// newModel M_RateLimiter
translateModel(
"ZFlib.Blocks.NonLinear.Tests.M_RateLimiter")
// newResultTest
RunScript(
"../TestReferences/M_RateLimiter_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_RateLimiter.mat
// endTest
```

Figure 8 *Modelica* Script built by the test collector

For each of these models, the collecting function searches for reference parameter files (exported from dsin.txt). These were saved earlier with the result reference files. For every parameter file that can be found, the model is simulated and tested against the corresponding reference result. The collecting of the tests leads to a *Modelica* script file that can be run by *Dymola*.

4.2 Performing tests

The test script is parsed by another *Matlab* script that calls the *Modelica* instructions sequentially and tests the results. Every line in this script is a test! First of all the whole package is syntactically checked. After this, each test model is translated. If this was successful, the simulation runs can be performed.

```
// preparation

// newModel M_StarterPrimitive
translateModel(
"ZFlib.Engines.Tests.M_StarterPrimitive")
// .. Caused an Error
// endTest
// newModel M_EngineTableWithBrake
translateModel(
"ZFlib.Engines.Tests.M_EngineTableWithBrake")
// newResultTest
RunScript(
"../TestReferences/M_EngineTableWithBrake_Ref
.mos")
// .. Caused an Error
// endTest
// newModel M_Clutch
translateModel(
"ZFlib.Mechanics.Rotational.Tests.M_Clutch")
// newResultTest
RunScript(
"..\TestReferences/M_Clutch_Ref.mos")
simulate
$copy  dsres.mat ..\temp\M_Clutch.mat
// endTest
// COMPARE DIFF M_Clutch   Correlation Coef-
ficient <0.99s.mat ..\temp\M_KK_3.mat
// endTest
```

Figure 9 Result script with failed commands and error description

The testing environment can test against the result file and against the log file (dslog.txt). In the result file, only signals that end in the `Outport` called Reference are tested (see Figure 7). Every command that was performed successful is appended to a text file. In the same way every command that causes any error is stored in another file. Nevertheless, the whole test script is executed. After the run has been completed, the user can take a look at the test result log files and decide on further actions.

The result log files can be run in the same way as the first test script. The log file provides some explanations of the errors, which supports the user in doing the corrections.

5 Conclusions

The `ZFlib` library has been used successfully in several projects working with various simulation environments. The ASCII file parameterization is well accepted. With the modified `PowerTrain` library we can use a lot of components and save time for implementing ZF-own models.

The enhanced “discrete” *Dymola*-Block works fine and quite fast (also in combination with ZBF parameterization).

The automated testing needs some extra effort. But on the other side, errors can be found quicker and other users get a better understanding of the tested element.

6 References

[1] L. R. Petzold. A description of DASSL: A differential-algebraic system solver. In R. S. Stepleman, editor, Scientific Computing, pages 65–68, Amsterdam, 1983, North-Holland.