Improving Interoperability of FMI-supporting Tools with Reference FMUs

Christian Bertsch[1] Award Mukbil[2] Andreas Junghanns[3]

[1]Corporate Research, Robert Bosch GmbH, Germany Christian.Bertsch@de.bosch.com
[2]Dept. of Informatics, Clausthal University of Technology, Germany, Awad.Mukbil@tu-clausthal.de
[3]QTronic GmbH, Germany, Andreas.Junghanns@QTronic.de

Abstract

The Functional Mockup Interface (FMI) is more and more adopted by industrial users, increasing the pressure for higher quality and standard compliance of FMI supporting tools. The FMI cross check infrastructure was created to support tool vendors in their quest for quality improvements and to give users some measure of confidence in the tool quality. Currently it is up to the tool vendors which FMUs to submit there. For this reason the features tested in the FMI cross check are incomplete and interpretation of failures is difficult. While for FMI export there is the FMU compliance checker to test a wide variety of FMI features, no means are available today to prove standard compliance for FMI import. This will be overcome by adding reference FMUs to the FMI cross check, testing specific features of the FMI standard for standard compliance and giving detailed feedback, if an importing tool violates the standard. The paper describes the realization and the importance of reference FMUs.

Keywords: FMI, Reference FMUs, Compliance, Testing

1 Introduction

The Functional Mock-up Interface (FMI) is a tool independent approach for model exchange (ME) and co-simulation (CS) (Blochwitz et al. 2011, 2012), and on the way to become the industry standard for exchange of models and cross-company collaboration (Bertsch et al. 2014). Its main purpose is to share and reuse simulation artifacts among a wide variety of tools and environments, by putting the model specifications into a simple compressed file called Functional Mockup Unit (FMU). The FMU contains a model description in XML format, source written in C and/or binaries ready to run and optional components such as documentation, model logo, etc.

Even before the release of the first version of FMI, several modeling and simulation tools started supporting the FMI standard. According to the official website of FMI project, more than 80 simulation tools support FMI version 1.0 and more than 40 support version 2.0. Many automotive Original Equipment Manufacturers (OEM) have committed themselves to support FMI as exchange format for simulation models.

Because industrial users must rely on the results of FMI-based simulations, the maturity of FMI implementations comes into focus. For this reason, the FMI project has organized the FMI cross check (XC, 2014), where FMI exporting tools can upload test FMUs together with reference solutions as comma-separated values (CSV) files, and importing tools can run those FMUs and report the results. Once the results have been submitted, they are shown in the FMI cross check table at the FMI official website, which helps users to check which tools work well together and which vendors are serious in supporting FMI. In our experience, this has improved the quality and the maturity of FMI support of tools significantly.

The FMU compliance checker (FMU CC, 2016) is an open source software tool that was initiated by the FMI project and implemented by Modelon AB,contracted by the Modelica Association. Its intention is to check compliance of a given FMU with the FMI standard. Through this compliance checker, users can get reports about a wide range of problems that could arise from loading FMUs, which in turn play an important role in validating the tools that create (i.e. export) FMUs.

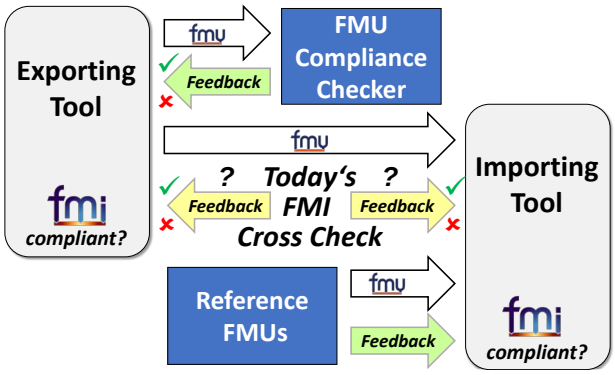


Figure 1: Three complementary ways of FMI compliance testing

According to the *"rule #8"* of the FMI cross check document (XC, 2014), vendors should test their FMUs using the FMU compliance checker before submitting

them, with or without reference results. This allows vendors to find problems in their implementations early. However, it is up to the exporting tool vendors which FMUs they submit to the FMI cross check, and (as depicted in Figure 1) in cases of problems it can be difficult to find out if it is a problem of the importing or exporting tool. This shows that special testing of FMI importing tools is no less important. We propose do realize this with the help of reference FMUs. Executing these in an importing tool can provide feedback on the FMI standard compliance of this tool as described below.

From the beginning, in the FMI cross check rules (XC, 2014) the important role of reference FMUs was foreseen. The contribution of this work is a step towards realizing such reference FMUs. In Sec. 2 we introduce the concepts, the requirements and the classifications of the reference FMUs. In Sec. 3 we present an initial implementation of reference FMUs. In Sec. 4 we present the means of testing the reference FMUs and first experience with FMI importing tools. Last but not least, we conclude our work and give an outlook to ongoing research in Sec. 5 respectively.

2 FMI and Reference FMUs

The FMI standard comes in textual form, supported by graphs, e.g., of the calling sequence, and XML schemata. (Blochwitz et al. 2012). The FMI 2.0 standard has added also a mathematical description of FMI, which clarifies a large number of concepts. However, the FMI standard is considered to be not fully formalized, which means the standard specifications are not written in formal description language. Formalizing the standard would help automatically generating test cases, and for validating FMUs statically or during runtime.

Formalizing the FMI standard has been partially addressed in some publications, e.g. (Hasanagić et al. 2016), but this seems far away from being realized for the whole standard in the next years. Thus, testing methods from software engineering come into the focus.

2.1 Reference FMUs and Software Testing

In order to test FMI standard compliance, we must test:

1. *Exporting tools:* these tools create FMUs.
2. *Importing tools:* these tools run FMUs.

The exporting tool should follow the FMI standard specifications for creating FMUs, such as providing a correct `modelDescription.xml` file, and use the correct naming and implementation for the functions as stated in the standard. The FMU compliance checker tests the validity of the FMUs and, implicitly, the exporting tools with respect to a large number of FMU properties. However, the FMU compliance checker can only check a finite number of FMU properties for correctness and is extended step by step. If the FMU compliance checker does not find a problem, it is not guaranteed that the FMU is error free.

In software engineering, the three basic types of software testing are (Bruegge, B., Dutoit, A. H., 2009):

- *Black-Box testing*: testing done by giving inputs and analyzing outputs. Tester does not use source code.
- *White-Box testing*: testing done with knowledge of the internals of the software.
- *Grey-Box testing*: a combination of the black-box and white-box techniques.

Testing FMUs using the FMU compliance checker is grey-box testing because there are open aspects about an FMU (like the `modelDescription.xml`) and closed aspects of an FMU (compiled dynamic link libraries (DLLs) containing the model behavior of the FMU). If the FMU comes with reference CSVs, then the FMU compliance checker sets the inputs according to the input CSV file, runs the FMU and compares the resulting outputs with the reference outputs.

Dealing with the importing tools is different. Most of the simulation tools supporting FMI are commercial, with unknown import mechanisms. Therefore, those importing tools are considered black-boxes, and the only way to test them is to run special FMUs that can spot problems and log errors. These special FMUs are called *"Reference FMUs"*.

2.2 Definition of Reference FMUs

"A reference FMU is an FMU specifically implemented to test compliance with a certain aspect of the FMI standard of a simulation tool. It has the ability to detect and log errors and wrong practices according to the FMI standard specifications. A reference FMU shall be inspected and reviewed before being accepted and published; thus, they must be available in source code and all the creation tools must be freely available."

This definition makes clear, that FMUs demonstrating a certain feature but exported by some commercial simulation tool cannot be considered as reference FMUs, because they might be too complicated, coming without the full source code and tools that are necessary to create them without having the needed licenses. However, they can also be very valuable. We encourage tool vendors also to export such "Feature demonstration FMUs" more often to the FMI cross check.

For the first set of reference FMUs we focus on testing "hard facts", e.g., testing standard compliance aspects of the importing tools such as data type support and correct calling sequences. Other goals such as testing usability of FMUs with many parameters or large input/output sets, simulation performance with many states or simulation performance with many algebraic/discrete equations are currently not covered.
We limit ourselves to "positive" test cases (that the importing tool should accept) and do not consider negative FMU cases (that should be rejected by the

DOI 10.3384/ecp17132533

importing tool), because we think that invalid or incorrect FMUs should be detected by the FMU compliance checker and in the FMI standard we have not seen requirements on an FMI importing tool to reject certain FMUs.

2.3 Requirements on Reference FMUs

Reference FMUs shall:

- test specific features of FMI importing tools, and the set of reference FMUs cover many features, and
- follow the FMI standard. A minimum requirement is, that the FMU compliance checker runs successfully (i.e., without warnings or errors) or that a trac issue has been created in case of limitations,
- be simple and well-documented,
- be of high quality; to this purpose they shall
- be reviewed according to defined rules
- be available with all source code and tools that are necessary to create them – in order that the creation process can be inspected and reproduced,
- detect and log the cause of a failure if possible, and
- fit into the FMI cross check infrastructure (if possible), e.g. by providing output signals.

The documentation of the reference FMUs is very important in order to reproduce the creation and interpret the results. It shall contain the following information:

- Authors, change history and review status of this reference FMU.
- The test purpose: What shall be tested with this FMU? Which potential errors of importing tools shall be detected with this FMU? Which capabilities of importing tools shall be tested?
- Implementation hints: How is this FMU created? (E.g. which libraries and tools are used?) Which steps or scripts have to be run to create the FMU?
- Test setup: What are inputs to this FMU (data type, values over time) and what are the expected outputs?
- Is this FMU suitable for the current FMI cross check infrastructure?

We have created a template for the documentation which will be made publicly available. The documentation will be contained within the reference FMUs as html documentation.

2.4 Sources of Reference FMUs and Coverage

There are several ways of deriving reference FMUs: One is to go systematically through the standard and trying to derive FMUs testing coverage and correct implementation of all features. Another is to implement FMUs based on (negative) experience with importing and simulating FMUs created by one and run in some other (presumably buggy) tool. For creating a reference FMU based on this experience, one should abstract the missing feature or error of the importing tool to a simple example fulfilling the requirements listed above and triggering the erroneous behavior.

In the following we followed both concepts. In the current work, we did not intent creating a complete set of reference FMUs, but we consider this as a starting point that can be extended by developers and users once the reference FMUs will be released to the FMI project and to the public.

For certain aspects of the FMI standard, measures of coverage can be derived: E.g., we have created reference FMUs for all supported data types or we check the allowed function calls in all FMI states and have created reference FMUs that reach all of these states.

2.5 Classifications of Possible Reference FMUs

FMI standard has main features and specifications that should be followed, and from those features we propose this classification of reference FMUs:

- FMUs for testing data type's capability (one for each data type),
- FMUs having dependencies on binaries, e.g. DLLs, or other resources, e.g. CSV files,
- FMUs testing correct interpretation of the `modelDescription.xml` file (version string, GUID, model identifier… etc.),
- FMUs for testing access restrictions depending on variable attributes (i.e. causality/variability combination),
- FMUs for testing the calling sequence as specified in the finite-state machine of the standard document,
- FMUs testing correct event handling (e.g., plausible event localization),
- FMUs for testing optional capabilities, e.g., partial derivatives, and
- Complex FMUs enabling the testing of the interactions of different features (e.g., having continuous states, multiple variables with different attributes, different kind of events).

The first FMUs in this list can be considered as single-feature "*diagnostic FMUs*", i.e., they are designed to test for a specific feature. A failure of them is very easy to interpret. It is intended that the features to be tested by different diagnostic FMUs are "orthogonal" in the sense that the feedback is as clear as possible.

On the other hand, the more complex "*multi-feature*" FMUs enable the detection of more subtle errors that only occur due the interplay of different effects or due to high complexity of the FMUs.

While in principle there could be completely different reference FMUs for ME and CS, for our first set of

reference FMUs we have created all our reference FMUs for both ME and CS.

2.6 Feedback of Reference FMUs

If an importing tool does not support a feature (e.g., specific data types) it should check this during FMU import based on the information in the `modelDescription.xml` file, and give a meaningful feedback: this is an "announced incompatibility".

During runtime – especially for diagnostic FMUs – an internal check of the reference FMU will trigger an FMI error, so that the simulation is stopped and a meaningful log message is created. Another possibility is, that the FMU runs without an error, but the outputs of the FMU are wrong. This can be detected, e.g. by the FMI cross check infrastructure.

In the best case the reference FMU is simulated by the importing tool without any error and produces the correct outputs (within specified tolerances).

3 Implementation of Reference FMUs

3.1 Tools to Create our Reference FMUs

The FMU Software Development Kit (FMUSDK, 2014) is considered a good starting point for supporting FMI and implementing reference FMUs. The FMUSDK demonstrates the basic use of Functional Mockup Units (FMUs) as defined by the FMI version 1.0 and 2.0 specifications (FMUSDK, 2014) and implemented by QTronic and freely available in open source. The FMUSDK is suitable to create source code FMUs in a quite simple manner, and already contains many checking mechanism for the functions calling sequence. Therefore, reusing and adding to this implementation helps in creating the first reference FMUs. Furthermore, we used the FMUSDK scripts and libraries for building our FMUs.

For the first reference FMUs we concentrated on FMI 2.0 FMUs for Windows 64-bit binaries. Extending this to other platforms is discussed in Sec. 4.2.

3.2 Preliminary Set of Reference FMUs

The basic structure of our reference FMUs is the same as proposed by the FMUSDK (Figure 2).

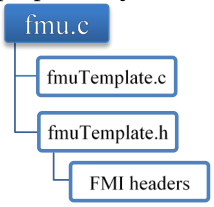


Figure 2: Reference FMU source code structure

The `fmuTemplate.c/.h` source files contain all necessary FMI function implementations and are included by the main FMU source file "`fmu.c`". We consider this structure versatile, because each FMU can reuse the template and just modify small code parts for the realization of specific features. As a starting point, we used the original template files from the FMUSDK. For advanced checks for the calling sequence, we modified the templates (see Sec. 3.2.5).

All of the source code is included in the FMU in order to enable inspection and debugging. Except for the FMU mentioned in 3.2.3, all FMUs are separately created as ME or CS FMUs. We will briefly describe our first set of reference FMUs:

3.2.1 FMUs for Testing Datatype Support

While the support for real variables is standard for FMI importing tools, the support for other data types is limited. String inputs/outputs are not standard in many simulation tools; however, it is expected that an importing tool gives a meaningful error message in such a case.

We created an FMU for testing of support for Boolean inputs/outputs (**bool.fmu**): We implemented a simple test of the Boolean data type support. This model demonstrates a simple AND gate logical operation, with two Boolean inputs, executing AND operation and a Boolean output with the result. This FMU uses three model variables: two Boolean inputs and one Boolean output.

Additionally, we implemented an FMU for testing of support for Integer inputs/outputs (**integer.fmu**): It implements the addition of two Integer inputs written to an output.

Further on, an FMU for testing String capability (**string.fmu**) was created: It gets a string input, concatenates it with a locally defined string and writes it to a String output. As stated in the FMI standard, the importer should provide his own allocating and freeing functions (e.g. `calloc`, `free`) along with the logger function. This property gives the importer the ability to manage memory also for the FMU. We use this allocation function to initialize strings, and problems of the importing tool arising from the allocation will be detected.

3.2.2 FMU for Testing FMI Version Number for Future Bugfix Release FMI 2.0.1 (ver.fmu)

This FMU tests if FMI 2.0 importing tools accept FMUs with a version string "2.0.1". A future version 2.0.1 of the FMI standard will have only clarifications about ambiguities in the FMI 2.0 standard. FMI 2.0.1 FMUs shall be valid FMI 2.0 FMUs (i.e., FMI 2.0 shall be "forward compatible") as mandated by the FMI development process (FMI DEV, 2015). This reference FMU contains no calculations.

3.2.3 FMU for Testing Events

We created an FMU with internal time events **tEvents.fmu**: it increments an internal integer variable every second; for ME, time events are defined for this

DOI 10.3384/ecp17132533

purpose; for CS the events are handled internally in the FMU.

State events are present by the bouncingBall example described in 3.2.6. Event handling shall be tested by more reference FMUs to be developed in the future, see Sec. 5.4.

3.2.4 FMU for Testing the Support of both ME and CS Support in One FMU (mecs.fmu)

This FMU contains both ME and CS binaries – without any calculations. The FMI 2.0 standard allows for having both ME and CS in one FMU. It is expected, that importing tools supporting only one of these FMI flavors, nevertheless accept such an FMU. This FMU was inspired by negative experience with one simulation tool only supporting ME import and rejecting FMUs containing ME and CS. This FMU is created with modified build scripts compared to the original FMUSDK, which can create only FMUs supporting either ME or CS.

3.2.5 Testing the Handling Additional Resources

We implemented an FMU testing the calling of an additional dynamic link library (DLL) in the binaries folder (**dll.fmu**): The FMI 2.0 standard allows the exporting tools to include additional binaries to be shipped along with the FMU. These libraries should be placed in the same folder of the compiled FMU binary (or binaries) and to test this capability we have created a simple FMU that contains and uses an additional DLL. This DLL is compiled for each specific target platform (i.e. 32-bit or 64-bit for Windows) with `/MT` option to include a run-time environment, which is also mentioned by the FMI standard when compiling the FMU source code. In our example, we included "`square.dll`", which includes a function that returns the square of a given real value. We use this function to calculate the square of an input and to write it to an output.

Additionally, we implemented an FMU shipped using a CSV file resource (**csv.fmu**): The FMI 2.0 standard enforced the importing tool to provide a clear, IETF RFC3986 compliant URI of resources location during FMU instantiation. According to the standard, this URI could be used for a local resources folder (prefixed by *'file:///'*) or for remote ones (prefixed by *'http:///'*, *'https:///'* or *'ftp:///'*). The resources folder is intended to be used *only* during FMU instantiation. To test this feature, we created a simple FMU that is shipped with a csv file in resources folder, which is accessed during instantiation. This csv file contains an integer value, and during instantiation we load this file, set the value stored in the csv file to a local variable and calculate the square of this value as an output.

3.2.6 FMUs for Testing the Calling Sequence

The FMUSDK has already implemented many checks regarding the calling sequence. However, we have gone through all states again and re-considered the allowed function calls. The FMI standard describes the calling sequences for ME and CS using finite-state machines and textual representations (Blochwitz et al., 2012). The finite-state machines and their legends also describe which functions are allowed in which state, including which categories of the variables are allowed to be accessed in each state, regarding the causality-variability-initial attributes.

The FMUSDK functions knows which state the FMU is in by an indicator and a table of allowed functions calls in each state is defined. The following features are already checked by the FMUSDK:

1. Whenever an FMI interface function is called, it checks whether the current state is among the allowed states, otherwise it returns `fmi2Error`. This ensures the detection of erroneous calling sequences.
2. `fmi2Instantiate` ≠ `NULL`. This can happen if:
   a. there is no valid logger function,
   b. no allocate/free function provided by importer,
   c. the `GUID` is inconsistent, or
   d. model variables did not initialize successfully.

Those features are clearly described in the state machine graphs. However, there are a few rules mentioned in the textual description of the FMI standard, that should also be checked, which are not yet handled by the FMUSDK. Those features are:

1. `Fmi2SetupExperiment` should be called at least once before `fmi2EnterInitializationMode`, although they can be called in the same state: instantiated.
2. `stopTime` and `Tolerance` are optional, but should be handled if set. If `stopTimeDefined = fmi2True`, then the independent variable time must not be set to a value greater than `stopTime.`
3. In case of CS, after an `fmi2SetXXX` call, there must be an `fmi2DoStep` before an `fmi2GetXXX` is allowed. In other words, the order `fmi2SetXXX - fmi2DoStep - fmi2GetXXX` must be followed.

Checks for these features are implemented in a modified version of the `fmuTemplate.h/.c` files. Several FMUs with an increasing difficulty using these modified template files have been created:

An FMU testing the correct calling sequence for an algebraic calculation for real variables has been implemented: (**real.fmu**): It sums two real inputs and writes them to the output.

An FMU testing the correct calling sequence for one continuous state was created (**dq.fmu**): This is a simple FMU with a continuous state, we used the Dahlquist's example from the FMUSDK.

Another FMU tests the correct calling sequence for continuous states and state events (**bB.fmu**): this is the BouncingBall example from the FMUSDK, which

contains two continuous states and state events. The simulation time for running this FMU shall be 2s, so that the phase of minimal amplitudes of the bouncing ball including the Zeno effect (Fritzson 2004) is currently excluded from the evaluation.

We used modified `fmuTemplate.c/.h` files with the additional checks.to create these FMUs.

3.2.7 Testing Variables Access Restrictions

One of the enhancements brought by the FMI 2.0 standard is the clear definition of variables access restrictions. The standard added more categories to the causality/variability attributes, and the "initial" attribute was added. According to the standard, a specific set of combinations are allowed to be used in describing model variables. Furthermore, there are restrictions in accessing model variables in each state according to the attribute combinations of the variables. These restrictions are considered to be part of the state-machine (Figure 3 and Figure 4), because not only a specific set of functions are allowed to be called in a state, but also a specific set of variables are allowed to be accessed in each state. E.g., discrete variables are only allowed to be accessed when an event is triggered. Another example is that the simulator should never set constant variables. A last example is, that continuous states may not be set via `fmi2SetReal`, but with `fmi2SetContinuousStates`. This is a basic idea of this kind of reference FMU and the authors are still working on these when submitting this paper.

Compared to the current implementation of the FMUSDK, the `template.c/.h` and the specific `fmu.c` files have to be enriched by additional information regarding the variable attributes, as the FMUSDK does not parse the `modelDescription.xml` during FMU creation and this information is currently not available to the implementation of the FMU during simulation.

4 Testing and Using Reference FMUs

We validated our reference FMUs with the FMU compliance checker, and tested them with several tools. This led to three tickets for clarification of the FMI standard version 2.0.1, three tickets for extension to the FMU compliance checker, several tickets for improvements of the FMUSDK and several bug reports regarding errors or improvements the tested tools.

4.1 Implementing an Erroneous Simulator

Two simple open source simulators come with the FMUSDK that import and run FMUs, one for ME and one for CS. We used these simulators to perform first test of the reference FMUs. Then we injected some faults (or wrong practices) in these simulators to check that these errors are detected by the reference FMUs and meaningful feedback is provided. Those faults are chosen carefully from our experience and from most frequent errors that occur. Examples of these faults are to initialize FMUs before instantiating, or to exit initialization mode before entering it. Another example for CS to call `fmi2SetXXX` and directly call `fmi2GetXXX` afterwards without an `fmi2DoStep` call between them.

4.2 Tests with Importing Tools - Overview

We tested the reference FMUs with 10 different tools for Windows 64-bit binaries. Most of these tools cope very well with normal FMUs generated by other simulation tools. However, with our reference FMUs we detect some limitations and bugs in the involved tools, which are communicated to the tool vendors and implementers.

In Table 1 and Table 2, we depict the result of our checks of the ME and CS Reference FMUs:

Table 1: Results for ME:

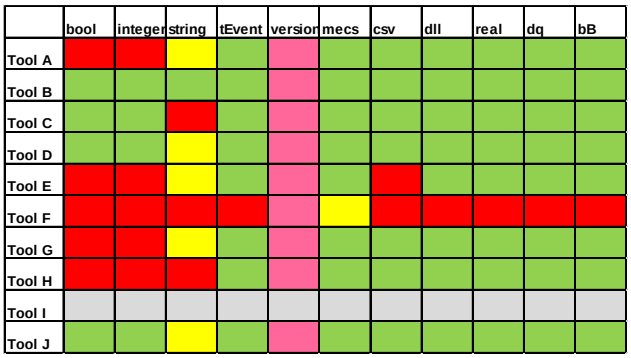
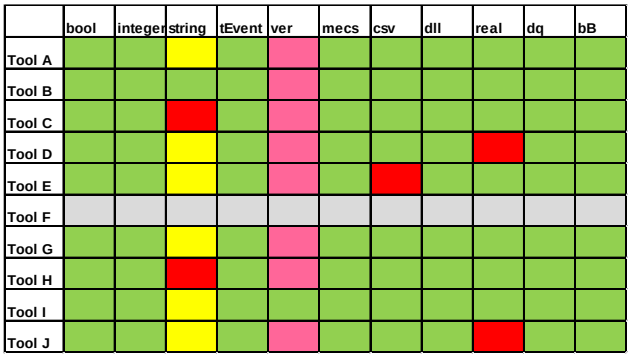


Table 2 Results for CS:



- OK
- "Announced limitation" of the tool
- Error or missing feedback for limitations
- Error; possibly standard clarification needed
- ME or CS not supported by the tool

Alone for testing the Windows 64-bit combinations of 10 tools with 11 ME and 11 CS FMUs, it took the effort of setting up and running more than 200 simulation models. Additionally, we tested some 32-bit tools, with no significant differences to the 64-bit results. The effort of running the tests and diagnosing the results will be shifted in the future to the tool vendors by including the reference FMUs in the FMI cross check (see Sec. 5.1).

DOI 10.3384/ecp17132533

In most tools the FMI import support has a quite mature quality level, and the problems encountered in our tests mostly are mainly due to our very strict diagnostics.

4.3 Problems Detected in Importing Tools

The reference FMUs for data type support revealed problems of several tools for non-real data types: For the FMUs testing for Integer and Boolean input data support (**bool.fmu** and **integer.fmu)**, five tools show errors in ME due to violations of the calling sequence: They want to set discrete values in continuous time mode, which is forbidden. (Remark: this is not an incompatibilty between hybrid modeling in Modelica and limitations of the Modelica standard; several Modelica-based tools do not have a problem with these FMUs).

The results for the **string.fmu** reflect, that String inputs and outputs are not supported by typical block-oriented simulation tools. However, it is expected that in this case, the tools give a meaningful diagnostic message during FMU import and not just ignore the string in/outputs.

The time events FMU **tEvents.fmu** was successfully run by all except one tool (violating the calling sequence).

A "2.0.1" version string in the **ver.fmu** as foreseen in the FMI 2.0 standard for a future FMI 2.0.1 FMU is problematic for all but one tested importing tools. FMUs following a future FMI 2.0.1 bugfix release, shall be valid "FMI 2.0" FMUs, see (FMI DEV, 2015). Thus, we suggest not to use the "2.0.1" version string in FMI 2.0.1, but "2.0". FMUs could contain the information that they follow the FMI 2.0.1 standard in an annotation in the `modelDescription.xml` file. This shall be discussed in the FMI project and clarified for FMI 2.0.1.

Only one tool is not able to import an FMU with both ME and CS support contained (**mecs.fmu**), but gives a meaningful feedback.

The **csv.fmu** crashes for one tool both in ME and CS.

The **dll.fmu** detects in one tool a violation of the calling sequence which is not related to DLL-access.

In ME, in none of the tools, except one, problems due to the additional checks in the calling sequence (**real.fmu, dq.fmu** and **bB.fmu**) have been detected. For CS, two tools violate the rule, that there may not be a call to `fmi2GetXXX` directly after an `fmi2SetXXX` without an `fmi2DoStep` call in between for **real.fmu**.

5 Outlook

The implemented reference FMUs will first be internally shared within the FMI project, e.g. within the "sandbox" of the FMI cross check infrastructure. The intention is to gather feedback both on the concept and the reference FMUs, to fix bugs and extend the documentation and to give tool vendors the opportunity to fix their implementations. The cross check working group of the FMI project will review the FMUs and discuss the proposed requirements on reference FMUs.

5.1 Usage within the FMI Cross Check

After the feedback and review phase within the FMI project we plan to publish the reference FMUs on the public part of the FMI project's resources on GitHub.

After acceptance by the FMI project, the part of the reference FMUs that fit into the FMI Cross Check infrastructure (e.g., w.r.t. to real inputs/outputs) shall be committed there and treated as an exporting tool and extensions to the infrastructure shall be considered.

5.2 Versioning and Indexing

When releasing the reference to the public within the FMI cross check, we will use a version number to refer to this release of the reference FMUs.

With the first release, we will propose an indexing of the FMUs by a naming convention enabling for a serial execution of the reference FMUs in a meaningful order. E.g. single-feature diagnostic FMUs should be performed before complex FMUs, so that the localization of errors is simplified. In this ordering, as few features as possible shall be added from one FMU to the next.

5.3 Extension of Supported Platforms

With the current solution, it is very easy to create reference FMUs for Windows 32-bit and 64-bit binaries. In order to support other platforms like Linux (32-bit/64-bit) or MAC OS X, the port of the FMUSDK to Linux (FMUSDK Linux, 2015) could be used. However, this version is not up to date with the latest version for the FMUSDK. Linux and OS X versions of the FMUSDK would be beneficial.

5.4 Increasing the Coverage

The first set of reference FMUS shall be extended by additional diagnostic and complex FMUs, e.g.:

- for systematically testing all kinds of events (state, time, externally triggered, zero crossings),
- dealing with a larger number of states (e.g. >=10 states with multiple events), and
- testing for optional capabilities of FMUs (e.g., partial derivatives).

Additionally, reference FMUs shall be implemented as proof of concept of new features of a future FMI standard from FMI Change Proposals (FCPs).

5.5 Connected FMUs and Parameter Sets

We also propose to extend the FMI cross check and reference FMUs to connected FMUs: for this purpose one could use connected FMUs inspired by the FMI 2.0 test FMUs (Test FMUs FMI 2.0 ME). However, these FMUs are implemented in Modelica, and need a Modelica tool for the generation of the C-code. Thus, it

would have to be clarified, if this can be done with a publicly available tool chain fulfilling our requirements listed in 2.4 or a re-implementation in C-code is needed. The definition of connected FMUs should be realized using the future System Structure and Parameterization (SSP) standard for the definition of connected FMUs (Köhler et al., 2016). Additionally, the SSP standard could be used to test the correct setting of parameter values to an FMU by the importing tool. For this purpose, the FMI cross check will have to be extended for connected FMUs.

5.6 Additional Benefit of Reference FMUs

Reference FMUs can also provide additional example implementations of FMUs to the FMI community that can serve as a starting point to implement FMI features in a good way; in other words, they give hints to the exporting tools of how typical FMUs could be implemented. This could help, e.g., for the handling of additional resources (binaries or other files), where we have observed many problems of tools in the past.

Reference FMUs can also contribute to clarifying unclear points of the FMI standard, as demonstrated e.g. for the version string reference FMU.

Additionally, with reference FMUs we can provide test FMUs for features that are not yet supported by (many) exporting tools, e.g. string inputs, provision of partial derivatives, and serialization of states.

6 Summary

In the current paper, we present the concept for the creation and usage of reference FMUs. As a starting point, first reference FMUs have been implemented and tests with importing tools have been performed, which led to the detection of several bugs in importing tools. This is seen as a proof of concept for the idea of reference FMUs. They shall be made publicly available first within the FMI project and then publicly by including them in the regular FMI cross check. With FMI community effort, the set of reference FMUs and thus feature coverage shall be increased. This will contribute to the improvement of quality of FMI importing tools.

Acknowledgements

The authors want to thank the members of the FMI cross check working group for their valuable input to our work.

Many thanks also to Torsten Blochwitz (ESI group), Umut Durak (DLR Braunschweig), Dan Henriksson (Dassault Systems), Adrian Tirea (QTronic), Karl Wernersson (Dassault Systems) for their valuable input and fruitful discussions.

Referenced Tools and Online Documents

(XC, 2014) FMI Cross Check Rules, v3.1, Modelica Association Project FMI: How to Improve FMI Compliance, June 2015. [Accessed online on Dec 16th 2016] https://www.fmi-standard.org/tools

(FMI DEV, 2015) FMI Development Process and Communication Policy, [Accessed online on Jan 22nd 2017] https://www.fmi-standard.org/development

(FMI 2.0 Standard, 2014) Functional Mock-up Interface for Model Exchange and Co-Simulation, Version 2.0, [accessed on Jan 21st 2017] https://www.fmi-standard.org/downloads

(FMU CC, 2016) FMU Compliance Checker, Modelon AB, released by Modelica Association Project FMI [accessed on Dec 16th 2016] https://www.fmi-standard.org/downloads

(FMUSDK, 2014) FMUSDK 2.0.4, QTronic GmbH, July 2014. [Accessed online on Dec 16th 2016] https://resources.qtronic.de/fmusdk/

(FMUSDK Linux, 2015) FMUSDK port to Linux and OS X, [accessed on Jan 6th 2017] https://github.com/cxbrooks/fmusdk2

(Test FMUs FMI 2.0 ME) Testing FMI 2.0 Model Exchange features of connected FMUs, Martin Otter, DLR, [Accessed on Jan 13th 2017] https://www.fmi-standard.org/downloads

References

Bertsch, C., Ahle, E., Schulmeister, U., The Functional Mockup Interface - seen from an industrial perspective, In: Proceedings of the 10th International Modelica Conference 2014, Lund, Sweden

Blochwitz, T., Otter M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.-V, Wolf, S., The Functional Mockup Interface for Tool independent Exchange of Simulation Models, In: Proceedings of the 8th International Modelica Conference 2011, Dresden, Germany

Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A., The Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models, In: Proceedings of the 9th Modelica Conference 2012, Munich, Germany

Bruegge, B., Dutoit, A. H., Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd edition, Prentice Hall Press, 2009, Upper Saddle River, USA

Fritzson, P., Principles of Object-Oriented Modeling and Simulation with *Modelica* 2.1, Wiley, 2004, Hobiken, USA

Hasanagić, M., Tran-Jørgensen, P. W. V., Lausdahl, K., Larsen, P. G., Formalising and Validating the Interface Description in the FMI Standard, FM 2016: Formal Methods, 2016, Springer, Heidelberg, Germany

Köhler, J., Heinkel, H.-M., Mai, P., Krasser, J., Deppe, M., Nagasawa, M., Modelica-Association-Project "System Structure and Parameterization" – Early Insights, Modelica Conference Japan, 2016

Pressman, R. S., Software engineering: a practitioner's approach, seventh edition. Publisher McGraw-Hill Higher Education, (2010), New York, USA