Application of Model-Based Testing to Dynamic Evaluation of Functional Mockup Units

Cláudio Gomes[1] Romain Franceschini[1,4] Nick Battle[2] Casper Thule[3] Kenneth Lausdahl[3]
Hans Vangheluwe[1] Peter Gorm Larsen[3]

[1]University of Antwerp, Belgium,
{claudio.gomes,romain.franceschini,hans.vangheluwe}@uantwerp.be
[2]Independent, United Kingdom, nick.battle@acm.org
[3]Aarhus University, Denmark, {casper.thule,lausdahl,pgl}@eng.au.dk
[4]University of Corsica, France

Abstract

A successful co-simulation standard is crucial in applying co-simulation in large scale distributed development processes. A factor that affects the success of a standard is how easily a vendor can implement it. In this paper, we describe an approach to facilitate the implementation of the Functional Mock-up Interface standard. In particular, we propose the use of model-based testing for evaluating the tools that export Functional Mock-up Units (FMUs). This has the benefit that the model used as documentation to describe the possible behaviors of an FMU, can also be used to test it. These principles are embodied in a tool, which is open source, and available online. We then use this tool to evaluate the FMUs available in the FMI Cross-check repository.

Keywords: model-based testing, functional mock-up interface standard, co-simulation

1 Introduction

Multi-paradigm modeling is a natural response to the challenges posed by the development of complex systems (Vangheluwe; Vangheluwe et al.). These challenges arise not only from essential system complexity (e.g., many interacting, heterogeneous, components), but also from the ensuing development process (e.g., concurrency and distribution) (Tomiyama et al.). Model integration is the means by which multiple models, constructed in different tools and formalisms, can be integrated to answer questions about the system these models represent.

Co-simulation is a technique where the models are integrated through their corresponding simulators (Kübler and Schiehlen, a,b; Gomes et al., f; Hafner and Popper; Palensky et al.). Each simulator, given inputs to the model, is capable of producing outputs, both function over time. Therefore simulators cooperate in producing the overall behavior of the system.

While co-simulation can only be used to answer questions about a system's behavior, it has the advantage that the contents of each model need not to be disclosed, as the model and solver can be encapsulated in a black box. It is therefore a suitable technique to address the challenges arising from concurrent and distributed development processes, where many tools/formalisms might be used and external suppliers may play a role.

Co-simulation standards are crucial enablers. These prescribe the interfaces with which inputs/outputs/parameters can be set/obtained and, optionally, the interaction protocol that each simulator abides to. For example, the Discrete Event System (DEVS) specification that prescribes the integration protocol between simulators (Zeigler; Gomes et al., f; Van Tendeloo and Vangheluwe). On the other hand, the Functional Mock-up Interface (FMI) Standard for co-simulation (FMIv2.0; Blochwitz et al., a,b) prescribes the interfaces, but under-specifies the interaction protocol. In this paper, we focus on the FMI version 2.0. In the FMI terminology, the simulators are referred to as Functional Mock-up Units (FMUs).

Past research (Schweiger et al., a,b), and the co-authors' experience, have shown that there are some ambiguities in the FMI standard (see, e.g., Section 5.2), which lead to not only technical difficulties (e.g., co-simulations crash), but also numerical difficulties (e.g., instabilities. For instance, the second and third most eminent barriers in the adoption of the FMI standard are: "Lack of transparency in features supported by FMI tools" and "insufficient documentation and a lack of examples, tutorials, etc." (Schweiger et al., b). Other somewhat barriers include: "It is difficult to implement FMUs", and "There is a lack of tools that sufficiently support FMI" (Schweiger et al., a). In the same empirical study, the authors identified the most experienced issue to be "Difficulties in practical aspects, like IT-prerequisites in cross-company collaboration."

Goal. We aim at supporting the community in rooting out possible ambiguities and improving the conformance to the standard. We propose an approach to the development of an evaluation tool for a co-simulation standard. Ideally, such a tool would never be necessary, as the ideal standard would allow for automated synthesis of co-simulation interfaces. However, we recognize that it is difficult to rigorously specify a standard to the level required by automated synthesis. In particular, we propose the use

of Model-Based Testing (MBT) for the evaluation. This has the benefit that the model used to describe the possible behaviors of a simulator can also be used to test it, and can be applied with minimal setup. We describe a tool that embodies these principles and tests co-simulation Functional Mock-up Units (FMUs) exported by tool providers. The tool does not test the numerical performance of an FMU (e.g., numerical error, freedom from stability problems, etc.). This tool is open source and available online with documentation and examples (Gomes et al., 2019). It is our goal that it helps in the development of FMUs and that our approach inspires standardization bodies to adopt the same technique.

Structure. In the next section, we survey prior work. Then, in Section 3, we describe our contribution. In Section 4 we describe the application of our contribution to empirically evaluate 169 FMUs, downloaded from the FMI Cross-check repository (Modelica Association, 2019a). Section 5 summarizes the results, lessons learned, and discusses the limitations of our approach. Section 6 concludes.

2 Related Work

Our goal is similar to other researchers that have helped identify ambiguities and omissions in the FMI standard.

In particular, the FMI development committee makes available an FMU Compliance Checker on its website (Modelica Association, 2019b). It verifies the consistency of the FMU metadata, and runs a co-simulation with user-defined input data using a fixed communication step size, appropriate to the FMU under test. This tool, along with the FMU-SDK (qTronic GmbH, 2019), available at the FMI web site, play an important role in improving the conformance of FMUs to the FMI standard. Moreover, the work in (Bertsch et al.) focuses on the testing of FMU importing tools, by the use of reference FMUs, build using the FMU-SDK.

Battle et al. has produced a tool that focuses on the conformance of the FMU metadata. It has been applied to the FMI Cross-check repository (Modelica Association, 2019a), and has yielded important lessons:

- Roughly 17% out of 692 FMUs do not follow the rules regarding the InitialUnknowns field of the model structure (FMIv2.0, Section 2.2.8, p. 60).
- About 18% have inconsistencies related to the derivatives declared. For instance, the derivative indexes do not match the set of real scalar variables with a derivative field declared; there are derivatives, but no real scalar variables with a derivative field declared; or there are real scalar variables with derivative field set, but no derivatives declared.

Our work aims at complementing the existing work by modeling all possible interactions with an FMU allowed by the FMI standard. In this sense, it obviously differs from the tool described in Battle et al.. However, our approach also differs from the FMU Compliance Checker, which runs a co-simulation with inputs prescribed by the FMU. While our tool also uses the information declared by the FMU, it may also stop a co-simulation, reset the FMU, initialize some variables while leaving others uninitiated, etc, as long as these operations are allowed by the standard.

Finally, we highlight the DCP-Test-Generator tool, made available in (Leibniz University Hannover, 2019). This tool was created with the same goal as ours, except it targets the Distributed Co-simulation Protocol (DCP) standard (Baumann et al.; Krammer et al., b,a). The DCP standard focuses on Hardware-in-the-loop and real-time co-simulations. It complements the FMI standard by encompassing information that is crucial for communication over a variety of transport protocols. At the heart of the standard is a state machine that dictates how the co-simulation progresses. This state machine was reused to generate test cases, providing anecdotal evidence that our approach has the benefit of leveraging any state machine diagrams used for documentation, leading to an inexpensive way of adopting any standard.

3 Model-Based Testing FMUs

In this section, we introduce Model-Based Testing (MBT) and then we explain the language we created to develop the FMU model, used in the MBT process.

3.1 Model-Based Testing

We adopt the terminology in Roy Awedikian and define Model-Based Testing (MBT) as the use of a model of the System-Under-Test (SUT) in order to guide test case generation. Such model can be represented as a state machine. Figure 1 shows an example that captures some of the possible interactions with an FMU.

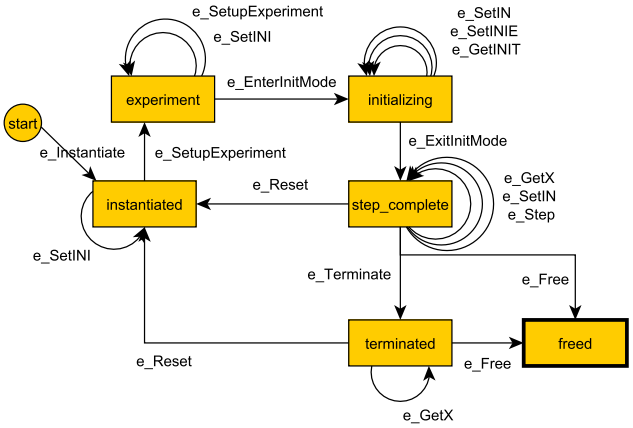

Figure 1. Example state transition system – Simplified FMU interaction model. The initial state is called "start", represented as a circle, and the final state is "freed" (outlined rectangle).

Any finite path accepted by the state transition system constitutes a possible test case of the SUT, and each test case must be associated with an oracle that dictates whether the SUT has passed the test or not. For instance, in Figure 1, an accepted path could be:

1. `Start-e_Instantiate->instantiated`
2. `instantiated-e_SetupExperiment->experiment`
3. `experiment-e_EnterInitMode->initializing`
4. `initializing-e_ExitInitMode->step_complete`
5. `step_complete-e_Free->freed`

where each edge corresponds to procedures that invokes the corresponding operations on the FMU. Some of these operations are detailed in Section 3 and all are available in the online code (Gomes et al., 2019).

If a test case fails, the MBT tool cleans up allocated resources, records the log of the SUT, and provides enough information to reproduce the test case (e.g., a seed value).

MBT is applied to SUTs as a black box, i.e., affecting only the SUT's inputs, making an ideal technique to test black box FMUs, and there is a large body of research on how to optimize the generation of paths to discover problems quickly (Li et al.; Peleska).

We have extended the Modbat (Artho et al., 2019; Artho et al.) to perform MBT on a finite state machine-based language that we developed. The language uses the GraphML syntax (Brandes et al.), for which the graph editor yEd (yWorks, 2019) can be used.

In the following sub-section, we describe the syntax and semantics of the proposed language.

3.2 Simulator Environment Language

We propose a language to describe all possible simulator operations. Our language is largely based on non-deterministic Extended Finite State Machines (EFSM), with constructions that make it easier to deal with large numbers of possible test paths.

Figure 1 shows an example of a traditional EFSM. Albeit a simple example, one can readily see the some of the problems associated with using a traditional EFSM for the description of simulator environments:

- There can be many edges between the same pair of states (e.g., edges from `step_complete` onto itself);
- There are operations that can be performed in almost all states (e.g., `e_Free` and `e_Reset`);
- There are many operations whose execution should not be repeated, or should be repeated a fixed number of times;
- Adding more interactions will quickly make the EFSM unreadable;

As a result, we propose the following extensions that are implemented in our language. These are purely syntactic, as they do not add to the expressive power of EFSMs, and their implementation can be done by reduction to a more complex traditional EFSM.

3.2.1 Edge-Or

This extension allows one to compactly represent multiple possible operations in the same edge. Its syntax and example reduction are described in Figure 2.

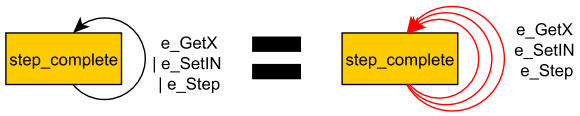

Figure 2. Edge-or syntactic extension and example reduction. Each edge-or (left) gets expanded to a traditional EFSM edge (right, in red).

3.2.2 State-Or

This extension allows one to compactly represent an operation between many possible states. Its syntax and example reduction are described in Figure 3.

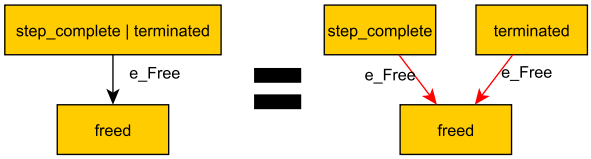

Figure 3. State-or syntactic extension and example reduction. Each State-or (left) gets expanded to a state (right), preserving the edges leaving/arriving at that state (in red).

3.2.3 Bounded Repetition

This extension allows one to compactly represent a self-loop edge that should be repeated only a finite number of times while in the same state. Its syntax and example reduction are described in Figure 4.

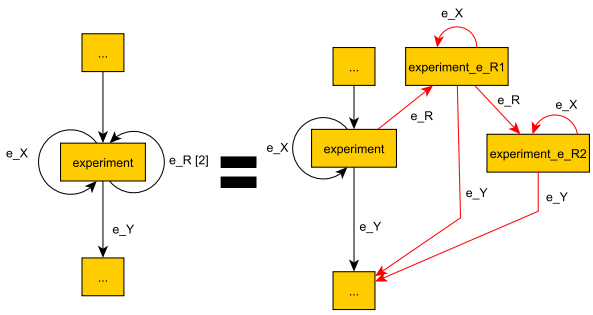

Figure 4. Bounded Repetition syntactic extension and example reduction. A bounded repetition edge (left) gets expanded to a counter of the number of times that edge has been executed (right). Notice that upon attaining the maximum number of edge executions, the state machine can only execute other edges.

3.2.4 Edge and State Merge

This extension, in combination with the previous ones, allows one to split the specification of the environment into multiple models. The merging of multiple descriptions is done by merging states with the same name, and taking the union of their edges. Figure 5 illustrates this operation. These reductions can be applied until none is applicable.

3.2.5 Edge Implementations

Each edge corresponds to a method, implemented in a class, as exemplified in Figure 6.

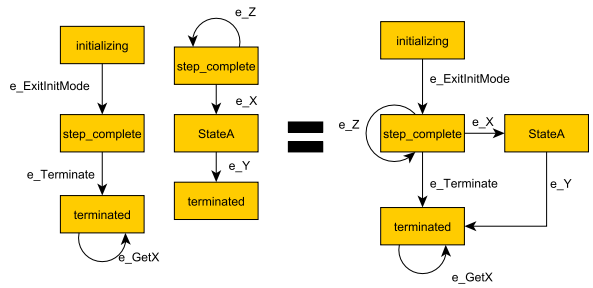

Figure 5. Edge and State Merge operation example. the two descriptions on the left get merged into one, on the right.

When setting/getting variables, we followed the definitions in (FMIv2.0, Fig. 11). For example, in Figure 6, the set `INIE` is defined as follows:

```
INIE = vars.filter(v =>
        v.causality==Causality.Input ||
        (v.variability != Variability.Constant
          && v.initial == Initial.Exact))
```

3.2.6 Tess Success Criteria

Whether a test passes or not is decided when the code corresponding to an edge is executed: any uncaught exception, including an assertion on a false statement, represents a test failure.

3.3 Model-Based Testing Tool for FMUs

The main use case and the structure of our tool are summarized in Figure 7.

4 Empirical Evaluation of FMUs

In this section, we describe the application of our tool to the FMUs made available for the FMI Cross-check repository (Modelica Association, 2019a).

4.1 Methodology

This goal of this study is to measure how well the FMUs tolerate sequences of operations that are valid with respect to the FMI standard.

The model used to generate test cases was created following (FMIv2.0, Fig. 11), and is partially illustrated in Figure 9. The full model, and edge implementations, can be consulted in the tool's repository (Gomes et al., 2019). Most operations are mapped directly to the FMI Standard operations. The following are the noteworthy exceptions:

- The `e_GetINIT` and `e_GetX` operations select at random one of the variables in the corresponding set `INIT` and `X`.
- The `e_SetINI`, `e_SetIN`, and `e_SetINIE`, operations select at random one of the variables in the corresponding set `INI`, `IN`, and `INIE`. The value to set the variable with is computed either from the declared nominal value, or is equal to 1.
- The `e_SetupExperiment` operation chooses, at random, whether the stop time, equal to 1s, is defined or not.

```
class FMIGraphModel extends ModBatGraphModel {
  // SUT and Time
  var instance: IFmiComponent
  var t = 0
  ...

  // Edge Methods
  def e_Instantiate() = {
    instance = instantiate(fmu, getGuid(fmu))
  }
  def e_SetINIE() = {
    // Pick a random var from the
    //  INIE set of variables
    // Pick a value (e.g., nominal value), and
    //  invoke the corresponding
    //  instance operation.
    setVar(getRandomElement(INIE))
  }
  def e_Terminate() = {
    val s = instance.terminate()
    assert(s == Fmi2Status.OK)
  }
  def e_Step() = {
    // Choose a step size according to
    // FMU Capabilities
    var H = chooseH()
    // Execute the step, and
    // check if the step was carried out
    val res = instance.doStep(t, H, true)
    assert(res == Fmi2Status.OK)
    t = t+H
  }
  def e_Free() = {
    instance.freeInstance()
  }
  ...
}
```

Figure 6. Pseudocode exemplifying implementation of some of the operations introduced in Figure 1. The full code is available online (Gomes et al., 2019).

- The `e_Step` operation will either pick a random step size (from the set $\{0.001, 0.01, 0.1\}$) if the FMU supports adaptive step size, otherwise 0.01 will be used.
- The `e_GetFmuState` and `e_SetFmuState` are only executed if the FMU supports the corresponding operations, and the state is set with the most recent state recorded.

Each operation is accompanied by a simple assertion checking whether it ran successfully. The sets of variables used are defined in Figure 8.

Our tool was applied to each FMU sequentially, with the following parameters: • Number of Random Walks=1000; • Self-Loop Execution Limit=10. The later refers only to the self-loops that are not already constrained by the bounded repetition operator. Both parameters are forwarded to Modbat (Artho et al., 2019).

For each FMU, the 1000 tests are executed in sequence. Before the first test is run, the FMU is loaded. Each test creates a new FMU instance and, regardless of the test result, that instance is always freed before the next test begins. After the 1000-th test is concluded, the FMU is

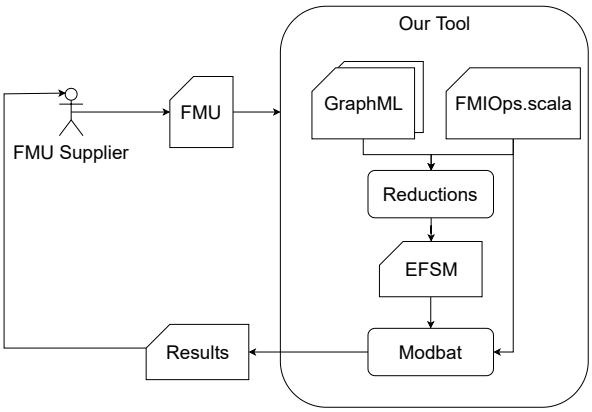

Figure 7. Main use case and structure of our tool. The FMU Supplier provides an FMU. The GraphML files, created with the simulator environment language (Section 3.2), along with the implementation of the edges, are part of our tool. These are loaded and transformed into a configuration that Modbat can use to run the MBT.

```
INI = vars.filter(v =>
  (v.variability != Variability.Constant &&
    (v.initial == Initial.Approx ||
     v.initial == Initial.Exact)) ||
  (ders.contains(v) &&
    v.`type`.`type` == Types.Real &&
    v.causality == Causality.Input))

IN = vars.filter(v =>
  v.causality==Causality.Input ||
  (v.causality != Causality.Parameter &&
    v.variability == Variability.Tunable))

INIE = vars.filter(v =>
  v.causality==Causality.Input &&
  (v.variability != Variability.Constant &&
    v.initial == Initial.Exact))

INIT = vars.filter(v =>
  v.causality==Causality.Output ||
  ders.contains(v) ||
  derMap.containsValue(v))

X = vars.filter(v => v.causality==Causality.Output)
```

Figure 8. Definition of the sets used in the get and set operations. The full code is available online (Gomes et al., 2019).

unloaded. This method avoids any concurrency problems within instances of the same FMU and different FMUs.

Each failed test is logged, along with the output log of the FMU instance and the sequence of operations that were executed from the instantiation until the failure.

Failed tests are grouped into equivalence classes. Two test failures are considered equivalent if they occurred in the same operation on an instance of the same FMU. For example, two tests failures on the `e_Reset` operation of an instance of the same FMU are considered equivalent, even if the first test failure happened just after the instance was instantiated, and the second test failure happened after a step was completed.

To analyze the results, we selected a test from each equivalence class and inspected the logs. The results are summarized in the next sub-section.

4.2 Results

The data used to write the results in this section can be obtained by contacting the authors. We do not mention any concrete FMU failures because we recognize that the responsible companies are working to improve their FMUs. Instead, we show aggregate statistics, and present a summary of the failures and the size of the respective equivalence class.

The aggregate statistics are: • FMUs passing all tests=77; • FMUs failing at least one test=55; • FMUs with process crash=37; • Total FMUs tested (sum of previous items)=169; • Total Tests=169000; • Total Failed Tests=14733; • Test Equivalence Classes/Analyzed Failures=102. When a process crash occurs, no other test on the FMU that caused the crash is run. Since this should never happen, we do not count such FMU as failing at least one test even though it did fail one test. Rather, we count it in the "FMUs with process crash" category.

The following is the list of the test failures:

Failure 1. The FMU does not recognize the value reference for a variable declared in its model description. Such a variable is set during initialization mode, and belongs to the set `INI`, as defined in Figure 8.

Failure 2. After an FMU is terminated, it fails when a variable belonging to the set `X` (Figure 8) is queried. Recall FMIv2.0, State "terminated", Fig. 11.

Failure 3. During stepping mode, a tunable parameter (i.e., a scalar variable with causality="parameter" and variability="tunable") is changed. The FMU then logs a message that it cannot be changed, and returns an error. Recall FMIv2.0, State "step Complete", Fig. 11.

Failure 4. The `getRealStatus` (invoked as part of the `e_GetLST` in Figure 9) operation is not supported after an instance is terminated. Recall FMIv2.0, State "terminated", Fig. 11.

Failure 5. URI has multiple possible formats for the absolute path of a file, and some FMUs only support one. This causes a failure in the instantiation of the FMU.

Failure 6. The outputs are queried after a change in the inputs, without a `doStep` in-between, causing the FMU to return an error. We investigate this issue in Section 5.

Failure 7. The `reset` operation is not supported.

Failure 8. Some FMUs do not isolate instances in the sense that one failed operation in an instance of an FMU will affect the outcome of other operation calls in a different instance of the same FMU.

Failure 9. A variable was set with a value that is outside the scope of an FMU.

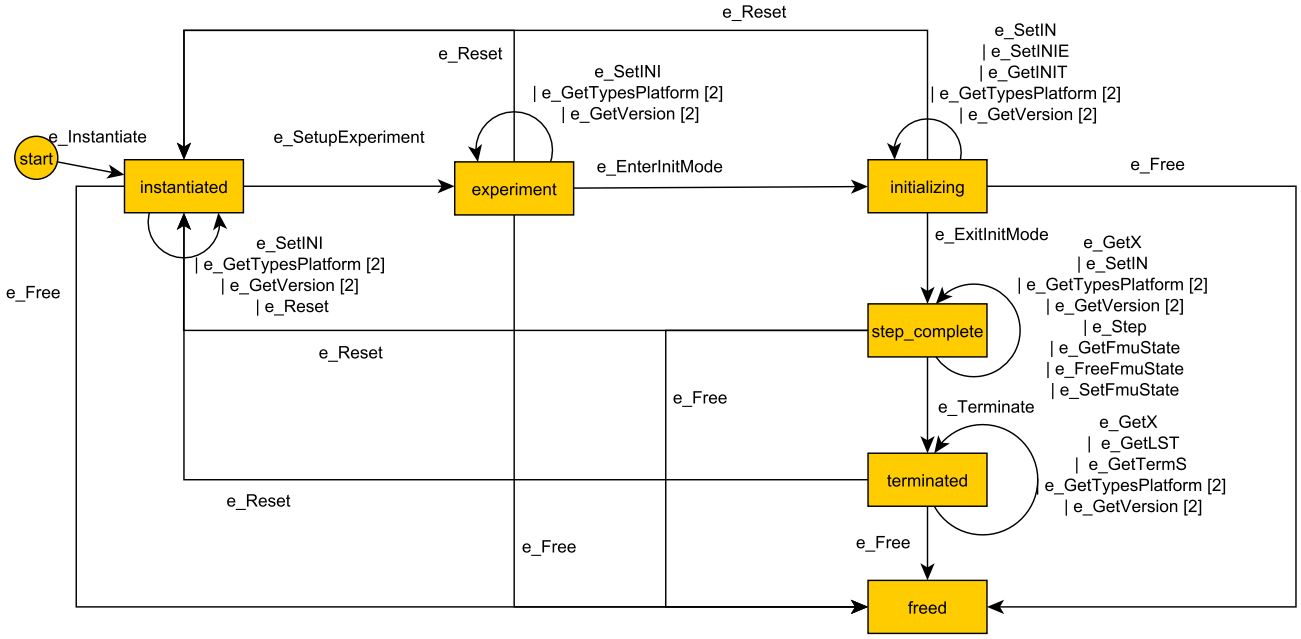

Figure 9. Partial model used to generate test cases. We did not consider asynchronous FMUs.

The majority of the process crashes are caused by invocation to the `reset` operation.

Figure 10 summarizes the number of occurrences of each failure, and relates some of them to the failures described above.

5 Discussion

In this section, we start by making explicit the limitations of this study, and how to mitigate them. Then, we discuss the lessons learned.

5.1 Limitations

An FMU that passes the tests provided in this tool does not constitute a proof that the FMU is correct. Currently, our tool uses the Modbat tool (Artho et al., 2019) to generate random walks in the graph, from start to finish. The later prints coverage information regarding the transitions taken by the model. We chose the number of tests per FMU to be 1000 because it allows us to cover about 95% of the possible choices in the model of Figure 9. Note that the coverage also includes the random choices made inside each edge operation (e.g., the choice of the scalar variable to set), as exemplified in Figure 6, but it depends on the number of scalar variables declared in each FMU. The self-loop limit is chosen to be a small value (10) because our goal is not to run co-simulations to the end. The performance of the tool was not a factor hindering our study. For example, on the example FMU that is shipped with the tool, it takes about 12 seconds to complete these tests.

An FMU failing a test does not necessarily mean that the FMU does not conform to the FMU standard. For example, it is acceptable that an FMU returns an error when the invocation of the `e_Reset` operation is made, but our tools will nevertheless signal a failed test. This enables us to measure the capabilities of the FMUs, which is important in the configuration of co-simulations involving those FMUs.

We have used Scala to implement our tool, and the INTO-CPS FMI library to load and interact with the FMUs. It is possible that some of the failed tests and/or process crashes are caused not by the FMU, but by the library. For the failures analyzed however, we did not find any evidence of this.

The choices made in the implementation of each operation are largely due to the co-authors' experience, and therefore can be a cause for error. Indeed, from the analyzed failures, only one crash was caused by the choice of parameters to run the co-simulation. The size of the equivalence class for this failure is 14 (out of 14733 tests). Whenever possible, we used the model description given by the FMU (e.g., default experiment, nominal/max/min values for variables). However, many FMUs do not provide such information.

Regarding the dataset used, it is worth noting that companies are continuously improving their FMI support and therefore these results will likely become outdated soon. However, the methodology and principles are embodied in an easy to use tool (Gomes et al., 2019).

Regarding the failure analysis, while there is some anecdotal evidence that test failures from the same equivalence class (as defined in Section 4.1) have the same cause, this was not exhaustively checked. Hence, it is possible that some test failures are not being reported here.

Regarding the model used for MBT, detailed in Figure 9, it does not yet cover all permissible operation sequences described in the FMI standard. In particular, we did not consider asynchronous stepping and state serial-

ization, assumed that state get/set operations can only be called while in the stepping mode, and that there's only one setup experiment (hence the `experiment` state). Moreover, a state based model may not be the best representation to specify more complex test scenarios. For example, a sound property of a rollback operation is that, under the same input signals, an instance that has been rolled back should have the exact same behavior as before the rollback. We are convinced that such property is more easily expressed in Linear Temporal Logic, rather than as a state machine.

Finally, we considered the FMI Standard version 2.0, and not the most recent (version 2.0.1), because, at the time of this study, most tools in the FMI cross check repository implement version 2.0.

5.2 Lessons Learned and Ambiguities

Despite the above limitations, we have extracted interesting insights and questions from these experiments.

Some FMUs have parameters that refer to numerical properties (such as internal solver step size). Since these are not model parameters, should they be standardized? If not, should the FMU avoid disclosing these parameters and, instead, attempt to infer appropriate values for these parameters, from the information provided by the master algorithm? There is some evidence that practitioners have difficulties configuring co-simulations (Schweiger et al., b), leading us to believe that fewer parameters translates to easier configuration. However, having such parameters exposed can enable the implementation of adaptive master algorithms (Gomes et al., b,a), or automated configuration techniques (Gomes et al., d; Holzinger and Benedikt)

Some FMUs have internal limitations on the values that variables can take, but fail to declare these constraints in their model description. When values on those variables violate these constraints, the FMU causes the co-simulation to fail, and the only way to know what caused the co-simulation to fail is to inspect the logs of the FMUs. Recognizing that there is a trade-of between demanding more standardization in error reporting, and having a wider adoption of such standardization, we recommend that the log messages of the FMUs be made clear regarding validity range violations.

We now focus on failure 6. We believe that such failure is a actually an intended feature of version 2.0 of the FMI standard (FMIv2.0). However, as we show next, this is not so clear, and only one tool reported failure 6. In particular, (FMIv2.0, Page 104) contains "There is the additional restriction in *slaveInitialized* state that it is not allowed to call fmi2GetXXX functions after fmi2SetXXX functions without an fmi2DoStep call in between". However, this is contradicted by the fact that the standard supports *feed-through* dependencies, which induce algebraic dependencies between inputs and outputs. To quote the standard:

- "*output*: The variable value can be used by another model or slave. The *algebraic* relationship to the inputs is defined via the dependencies attribute of `<ModelStructure><Outputs><Unknown>`.", page 45.
- "Attribute dependencies defines the dependencies of the outputs from the knowns [...] at the current Communication Point (CoSimulation).", page 58.

Since even the simplest mechanical systems, such as mass-spring-dampers, when coupled with a power bond in a co-simulation, exhibit such feed-through effect (Gomes et al., e), it is not clear that failure 6 is a failure of the standard.

In order to investigate how this ambiguity is being interpreted in practice, we devised a simple test, that follows the formal definition of feed-through: the input u_c feeds through to output y_c when there exist two different values v_1, v_2 and FMU state s_c that lead to two different output values (Gomes et al., c):

$$\texttt{get}_c(\texttt{set}_c(s_c, u_c, v_1), y_c) \neq \texttt{get}_c(\texttt{set}_c(s_c, u_c, v_2), y_c). \quad (1)$$

The test tries to find two values for which the above holds. If such values are found, the FMU is said to implement feed-through.

Out of 113 FMUs, 7 implement feed-through as defined in Equation (1). Moreover, from the authors' experience with the tool OpenModelica (Open Source Modelica Consortium, 2019), FMUs from the latest version (1.13.2) implement feed-through, whereas FMUs from the versions 1.12.X do not, further highlighting the uncertainty surrounding this feature.

Acknowledging that different versions of the same tool have different degrees of support for the FMI standard, we argue that the tools listed in FMI standard website should disclose which version is being used in the FMI Cross-check. This helps users to determine to which degree their version supports the standard.

Finally, we remark that, in the definition of the sets of variables `INI`, `IN`, `INIE`, `INIT`, and `X` (recall Figure 8 and (FMIv2.0, Figure 11)), the authors had difficulties understanding whether the `Set*` operations cannot be made on variables whose causality is output, and whether the `Get*` operation cannot be made on variables whose causality is input. For instance, the definition of `INIE`, is "any variable with variability $\neq$ 'constant' and with initial='exact'" (FMIv2.0, Figure 11), and does not explicitly state that causality='input' should be the case. Instead, elsewhere in the same page, "[In Initialization Mode] Variables with initial = 'exact' , as well as variables with variability = 'input' can be set" (FMIv2.0, Page 103). We suspect that failures 1 and 2 might be caused by misinterpretation of the definition of these sets.

6 Summary and Future Work

With the goal of rooting out possible ambiguities and improving the conformance to the FMI standard, we describe an approach to apply MBT to evaluate FMUs. We have contributed with a state machine based language that can be used to describe test cases for FMUs, and we package it in an easy to use tool, available online (Gomes et al.,

2019). We then used this tool to empirically evaluate existing FMUs and discussed the results. Our approach is easy to apply in existing and subsequent version of co-simulation standards. For the test failures observed, we are currently contacting the companies responsible, with the necessary information to ensure they can reproduce the failure.

In the future, we intend to extend the state machine that is used to generate tests. In particular, we will consider FMU soundness properties such as rollback and reset contracts, and step rejection (Broman et al.) and tunable parameter properties. Our language might have to be extended to specify such soundness properties. One possible extension is to implement linear temporal logic monitors. This will separate the property specification from the system model, and the former should guide the model-based testing of the latter.

Acknowledgments

We are grateful to the Research Foundation - Flanders (File Number 1S06316N) and to the Poul Due Jensen Foundation, for the financial support. Additionally, we thank the developers of the INTO-CPS Library and Modbat tools, on which our contribution builds upon. Finally, we thank Martin Krammer, Martin Benedikt, and the reviewers, for the information on the related work and throughout feedback on this work.

References

Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A Model-Based API Tester for Event-Driven Systems. In *Hardware and Software: Verification and Testing*, volume 8244, pages 112–128. Springer International Publishing. ISBN 978-3-319-03076-0 978-3-319-03077-7. doi:10.1007/978-3-319-03077-7_8.

Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat repository, 2019. `https://github.com/cyrille-artho/modbat`, accessed 25th September 2019.

Nick Battle, Casper Thule, Cláudio Gomes, Hugo Daniel Macedo, and Peter Gorm Larsen. Towards a Static Check of FMUs in VDM-SL. In *17th Overture Workshop*, page to be published.

Peter Baumann, Martin Krammer, Mario Driussi, Lars Mikelsons, Josef Zehetner, Werner Mair, and Dieter Schramm. Using the distributed co-simulation protocol for a mixed real-virtual prototype. In *2019 IEEE International Conference on Mechatronics (ICM)*, volume 1, pages 440–445. IEEE Industrial Electronics Society.

Christian Bertsch, Awad Mukbil, and Andreas Junghanns. Improving Interoperability of FMI-supporting Tools with Reference FMUs. pages 533–540. doi:10.3384/ecp17132533.

Torsten Blochwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *9th International Modelica Conference*, pages 173–184. Linköping University Electronic Press, a. doi:10.3384/ecp12076173.

Torsten Blochwitz, Martin Otter, Martin Arnold, C. Bausch, Christoph Clauss, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, M. Monteiro, T. Neidhold, Dietmar Neumerkel, Hans Olsson, J.-V. Peetz, and S. Wolf. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, pages 105–114. Linköping University Electronic Press; Linköpings universitet, b. doi:10.3384/ecp11063105.

Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. *Graph Markup Language (GraphML)*.

David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of FMUs for co-simulation. In *Eleventh ACM International Conference on Embedded Software*, page Article No. 2. IEEE Press Piscataway, NJ, USA. ISBN 978-1-4799-1443-2.

FMIv2.0. Functional Mock-up Interface for Model Exchange and Co-Simulation. URL `https://fmi-standard.org/downloads/`.

Cláudio Gomes, Romain Franceschini, Nick Battle, Casper Thule, Kenneth Lausdahl, Hans Vangheluwe, and Peter Gorm Larsen. FMIMOBSTER repository, 2019. `https://msdl.uantwerpen.be/git/claudio/FMIMOBSTER`, accessed 25th September 2019.

Cláudio Gomes, Benoît Legat, Raphaël Jungers, and Hans Vangheluwe. Minimally Constrained Stable Switched Systems and Application to Co-simulation. In *IEEE Conference on Decision and Control*, pages 5676–5681, a. doi:10.1109/CDC.2018.8619223.

Cláudio Gomes, Benoît Legat, Raphaël M. Jungers, and Hans Vangheluwe. Stable Adaptive Co-simulation: A Switched Systems Approach. In *IUTAM Symposium on Co-Simulation and Solver Coupling*, volume 35, pages 81–97. Springer, Cham, b. doi:10.1007/978-3-030-14883-6_5.

Cláudio Gomes, Levi Lucio, and Hans Vangheluwe. Semantics of Co-simulation Algorithms with Simulator Contracts. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 784–789. IEEE, c. doi:10.1109/MODELS-C.2019.00124.

Cláudio Gomes, Bentley James Oakes, Mehrdad Moradi, Alejandro Torres Gamiz, Juan Carlos Mendo, Stefan Dutre, Joachim Denil, and Hans Vangheluwe. HintCO - Hint-Based Configuration of Co-Simulations. In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 57–68, d. ISBN 978-989-758-381-0. doi:10.5220/0007830000570068.

Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art, e. URL http://arxiv.org/abs/1702.00686.

Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: A Survey. 51(3):Article 49, f. doi:10.1145/3179993.

Irene Hafner and Niki Popper. On the terminology and structuring of co-simulation methods. In Dirk Zimmer and Bernhard Bachmann, editors, *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 67–76. ACM Press. ISBN 978-1-4503-6373-0. doi:10.1145/3158191.3158203.

Franz Holzinger and Martin Benedikt. Optimal Trigger Sequence for Non-iterative Co-simulation:. In *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 80–87. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-381-0. doi:10.5220/0007833800800087.

R. Kübler and W. Schiehlen. Two Methods of Simulator Coupling. 6(2):93–113, a. ISSN 1387-3954. doi:10.1076/1387-3954(200006)6:2;1-M;FT093.

R. Kübler and W. Schiehlen. Modular Simulation in Multibody System Dynamics. 4(2-3):107–127, b. ISSN 1384-5640. doi:10.1023/A:1009810318420.

Martin Krammer, Martin Benedikt, Torsten Blochwitz, Khaled Alekeish, Nicolas Amringer, Christian Kater, Stefan Materne, Roberto Ruvalcaba, Klaus Schuch, Josef Zehetner, Micha Damm-Norwig, Viktor Schreiber, Natarajan Nagarajan, Isidro Corral, Tommy Sparber, Serge Klein, and Jakob Andert. The Distributed Co-Simulation Protocol for the Integration of Real-Time Systems and Simulation Environments. In *Proceedings of the 50th Computer Simulation Conference*, page No. 1. Society for Computer Simulation International, a. doi:10.22360/summersim.2018.scsc.001.

Martin Krammer, Klaus Schuch, Christian Kater, Khaled Alekeish, Torsten Blochwitz, Stefan Materne, Andreas Soppa, and Martin Benedikt. Standardized Integration of Real-Time and Non-Real-Time Systems: The Distributed Co-Simulation Protocol. In *Proceedings of the 13th International Modelica Conference*, volume 157, pages 87–96. Modelica Association, b. doi:10.3384/ecp1915787.

Leibniz University Hannover. Dcp-test-generator repository, 2019. https://github.com/modelica/DCPTestGenerator, accessed 25th September 2019.

Wenbin Li, Franck Le Gall, and Naum Spaseski. A Survey on Model-Based Testing Tools for Test Case Generation. In Vladimir Itsykson, Andre Scedrov, and Victor Zakharov, editors, *Tools and Methods of Program Analysis*, volume 779, pages 77–89. Springer International Publishing. ISBN 978-3-319-71733-3 978-3-319-71734-0. doi:10.1007/978-3-319-71734-0_7.

Modelica Association. FMI cross-check repository, 2019a. https://github.com/modelica/fmi-cross-check/tree/master/fmus/2.0/, accessed 25th September 2019.

Modelica Association. FMI standard website, 2019b. https://fmi-standard.org, accessed 25th September 2019.

Open Source Modelica Consortium. Openmodelica, 2019. https://openmodelica.org/, accessed 25th September 2019.

Peter Palensky, Arjen A. Van Der Meer, Claudio David Lopez, Arun Joseph, and Kaikai Pan. Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling. 11(1):34–50. ISSN 1932-4529. doi:10.1109/MIE.2016.2639825.

Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. 111:3–28. ISSN 2075-2180. doi:10.4204/EPTCS.111.1.

qTronic GmbH. FMU software development kit, 2019. https://github.com/qtronic/fmusdk, accessed 25th September 2019.

Bernard Yannou Roy Awedikian. *Practical Model-Based Testing*. ISBN 978-0-12-372501-1.

Gerald Schweiger, Cláudio Gomes, Georg Engel, Irene Hafner, Josef Schoeggl, Alfred Posch, and Thierry Nouidui. Functional Mock-up Interface: An empirical survey identifies research challenges and current barriers. In *Proceedings of the American Modelica Conference*, pages 138–146. Linköping University Electronic Press, Linköpings Universitet, a. ISBN 978-91-7685-148-7. doi:10.3384/ecp18154138.

Gerald Schweiger, Cláudio Gomes, Georg Engel, Irene Hafner, Josef-Peter Schoeggl, Alfred Posch, and Thierry Nouidui. An empirical survey on co-simulation: Promising standards, challenges and research needs. 95:148–163, b. ISSN 1569190X. doi:10.1016/j.simpat.2019.05.001.

T. Tomiyama, V. D'Amelio, J. Urbanic, and W. ElMaraghy. Complexity of Multi-Disciplinary Design. 56(1):185–188. ISSN 00078506. doi:10.1016/j.cirp.2007.05.044.

Yentl Van Tendeloo and Hans Vangheluwe. An Introduction to Classic DEVS.

Hans Vangheluwe. Foundations of Modelling and Simulation of Complex Systems. 10. doi:10.14279/tuj.eceasst.10.162.148.

Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AI, Simulation and Planning in High Autonomy Systems Conference*, pages 9–20. Society for Computer Simulation International.

yWorks. yed graph editor website, 2019. https://www.yworks.com/products/yed, accessed 25th September 2019.

Bernard P. Zeigler. *Theory of Modelling and Simulation*. New York, Wiley. ISBN 0-471-98152-4.

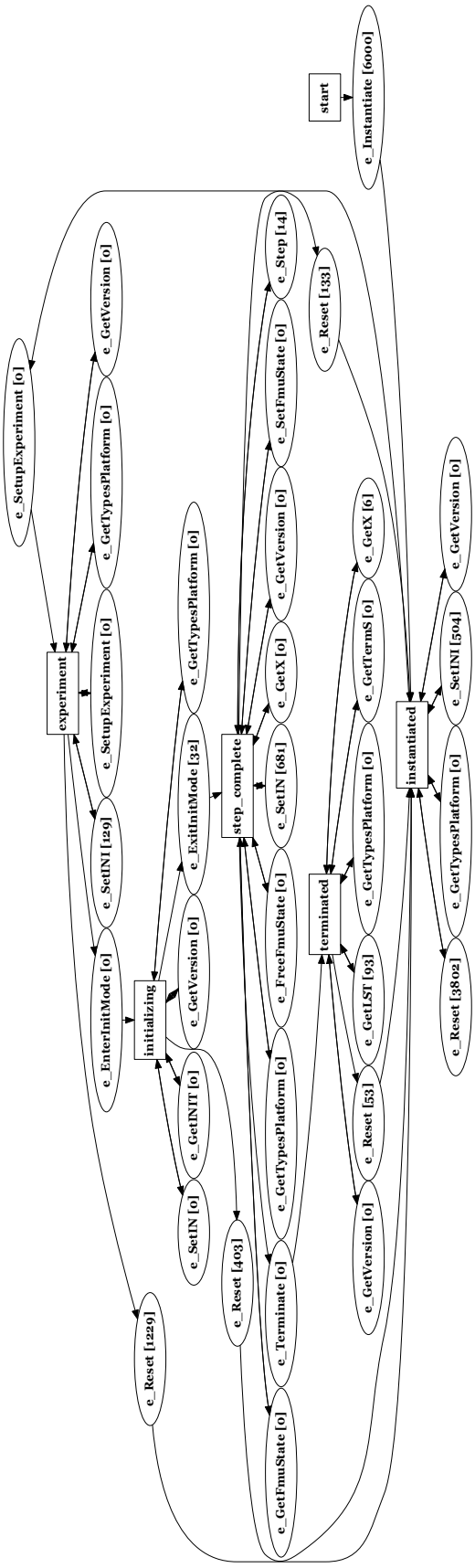

Figure 10. Number of occurrences of ea ch failure. Each failure leads to a trace. All traces were joined and counts were taken of the edges that caused the failure. The size equivalence class of each failure can be computed by summing the occurrences for each edge corresponding to an operation. Many tests failed at instantiation because of failure 5. Some of the test failures when entering and exiting initialization mode are due to failure 9.